



## PATENT ABSTRACTS OF JAPAN

(11) Publication number: **11212808 A**(43) Date of publication of application: **06.08.99**

(51) Int. Cl. **G06F 9/46**  
**G06F 12/00**  
**G06F 12/02**

(21) Application number: **10260427**(22) Date of filing: **14.09.98**(30) Priority: **21.11.97 JP 09321766**(71) Applicant: **OMRON CORP**

(72) Inventor: **HIRONO MITSUAKI**  
**INUI KAZUYUKI**  
**KONAKA YOSHIHARU**  
**KURIBAYASHI HIROSHI**

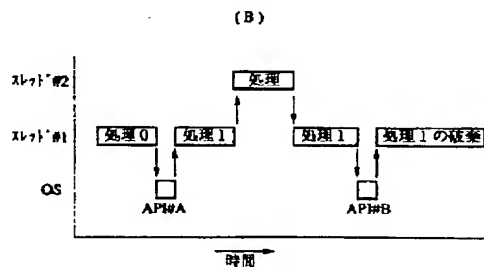
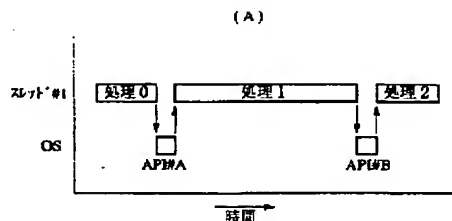
(54) **PROGRAM CONTROLLER, AND DEVICE AND METHOD FOR MEMORY ALLOCATION**

## (57) Abstract:

**PROBLEM TO BE SOLVED:** To secure real-time processing by guaranteeing the exclusiveness of a process without locking a system, to eliminates the need of compaction even by a mark & sweep method, to improve memory use efficiency, to reduce fragments in consideration of life for each object, and to reduce a waste of CPU power.

**SOLUTION:** An application program interface API#A which makes a request to start context switch generation presence/absence detection before the start of a process 1 is issued and an application program interface API#B which makes a request to end the context switch generation absence/presence detection at the end of the process 1 is issued. It is decided from the return value of this API#B whether or not context switching has been done halfway in the process 1 and when so, the process 1 is discarded.

COPYRIGHT: (C)1999,JPO



(19) 日本国特許庁 (J P)

(12) 特 許 公 報 (B 2)

(11) 特許番号

特許第3027845号  
(P3027845)

(45) 発行日 平成12年4月4日 (2000.4.4)

(24) 登録日 平成12年2月4日 (2000.2.4)

(51) Int.Cl.<sup>7</sup>

識別記号

F I

G 0 6 F 9/46

3 4 0

G 0 6 F 9/46

3 4 0 B

12/00

5 9 1

12/00

5 9 1

12/02

5 1 0

12/02

5 1 0 A

請求項の数 5 (全 42 頁)

(21) 出願番号 特願平10-260427

(22) 出願日 平成10年9月14日 (1998.9.14)

(65) 公開番号 特開平11-212808

(43) 公開日 平成11年8月6日 (1999.8.6)

審査請求日 平成11年2月15日 (1999.2.15)

(31) 優先権主張番号 特願平9-321766

(32) 優先日 平成9年11月21日 (1997.11.21)

(33) 優先権主張国 日本 (J P)

(73) 特許権者 000002945

オムロン株式会社

京都府京都市右京区花園土堂町10番地

(72) 発明者 廣野 光明

京都府京都市右京区花園土堂町10番地

オムロン株式会社内

(72) 発明者 乾 和志

京都府京都市右京区花園土堂町10番地

オムロン株式会社内

(72) 発明者 小中 義治

京都府京都市右京区花園土堂町10番地

オムロン株式会社内

(74) 代理人 100084548

弁理士 小森 久夫

審査官 久保 光宏

最終頁に続く

(54) 【発明の名称】 プログラム制御装置および方法

1

(57) 【特許請求の範囲】

【請求項1】 或るスレッドからの、コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しに応答して、コンテキストスイッチの発生有無を示すフラグをコンテキストスイッチが発生していない状態に設定する手段と、  
前記フラグがコンテキストスイッチの発生していない状態に設定された後、スケジューラによってコンテキストスイッチが行われたとき、前記フラグをコンテキストスイッチが発生した状態に設定する手段と、  
前記スレッドからの、コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しに応答して、前記フラグの状態に応じた値を前記スレッドに返す手段と、  
前記コンテキストスイッチが発生したとき、前記コンテ

2

キストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しから前記コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しまでの間の前記スレッドの処理を無効にする手段と、  
前記或るスレッドの優先度を高い状態と低い状態とに交互に変更するとともに、前記コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しから前記コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しまでの処理時間を受け取って、前記スレッドの優先度が高い状態のとき、該スレッドの優先度が低い状態になるまでの残時間と前記処理時間とを比較し、前記残時間が前記処理時間より短いことを検知したときに、前記スレッドの優先度を低くす

る手段とを設けて成るプログラム制御装置。

【請求項2】 前記スレッドは、メモリのヒープ領域内の他のいずれかのオブジェクトから参照されているオブジェクトを検出し、当該オブジェクトを前記ヒープ領域内の所定領域に複写する複写方式のガベージコレクションスレッドである請求項1に記載のプログラム制御装置。

【請求項3】 前記スレッドは、メモリのヒープ領域内のどのオブジェクトからも参照されないオブジェクトのメモリ領域を他のオブジェクトのメモリ割り当て可能なフリー領域として解放することにより生じるフラグメントを解消するメモリコンパクションスレッドである請求項1に記載のプログラム制御装置。

【請求項4】 或るスレッドからの、コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しにตอบสนองして、コンテキストスイッチの発生有無を示すフラグをコンテキストスイッチが発生していない状態に設定するステップと、前記フラグがコンテキストスイッチの発生していない状態に設定された後、スケジューラによってコンテキストスイッチが行われたとき、前記フラグをコンテキストスイッチが発生した状態に設定するステップと、前記スレッドからの、コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しにตอบสนองして、前記フラグの状態に応じた値を前記スレッドに返すステップと、前記コンテキストスイッチが発生したとき、前記コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しから前記コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しまでの間の前記スレッドの処理を無効にするステップと、前記或るスレッドの優先度を高い状態と低い状態とに交互に変更するとともに、前記コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しから前記コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しまでの処理時間を受け取って、前記スレッドの優先度が低い状態のとき、該スレッドの優先度が低い状態になるまでの残時間と前記処理時間とを比較し、前記残時間が前記処理時間より短いことを検知したとき、前記スレッドの優先度を低くするステップとから成るプログラム制御方法。

【請求項5】 或るスレッドからの、コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しにตอบสนองして、コンテキストスイッチの発生有無を示すフラグをコンテキストスイッチが発生していない状態に設定する処理プログラムと、前記フラグがコンテキストスイッチの発生していない状

態に設定された後、スケジューラによってコンテキストスイッチが行われたとき、前記フラグをコンテキストスイッチが発生した状態に設定する処理プログラムと、前記スレッドからの、コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しにตอบสนองして、前記フラグの状態に応じた値を前記スレッドに返す処理プログラムと、前記コンテキストスイッチが発生したとき、前記コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しから前記コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しまでの間の前記スレッドの処理を無効にする処理プログラムと、前記或るスレッドの優先度を高い状態と低い状態とに交互に変更するとともに、前記コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しから前記コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しまでの処理時間を受け取って、前記スレッドの優先度が低い状態のとき、該スレッドの優先度が低い状態になるまでの残時間と前記処理時間とを比較し、前記残時間が前記処理時間より短いことを検知したとき、前記スレッドの優先度を低くする処理プログラムとを記録して成るプログラム記録媒体。

【発明の詳細な説明】

【0001】

【発明の属する技術分野】この発明は、コンピュータのプログラム制御装置およびその方法に関する。

【0002】

【従来の技術】先ず、本願発明の明細書において用いる各種用語の定義を以下に示す。尚、出典記号は次のとおりである。

【0003】

JIS：JIS工業用語大辞典第4版（日本規格協会刊）

参1：情報技術用語事典（オーム社刊）

参2：情報処理用語大事典（オーム社刊）

参3：先端ソフトウェア用語事典（オーム社刊）

参A：SUPER ASCII Glossary Help on Internet

(1) プロセス(process)：処理や過程を意味する一般用語（参2）

(2) スレッド(thread)：プロセスまたはタスクと呼ばれる1つの実行環境の中で並列実行可能な処理を複数に分割した場合に、プロセッサのスケジュール対象となる実行単位または制御フローのこと。（参3）

(3) コンテキスト(context)：（オブジェクト指向システムに関連して）メソッドを実行するために必要な情報を蓄えるオブジェクトをいう。コンテキストは、呼び出し先のコンテキスト、プログラムの入ったメソッドオブ

ジェクト、プログラムカウンタ、スタックポインタという情報、引数や一時変数を取るところ、および評価スタックから成立する。このような実行環境をオブジェクトとしてとるが、プロセスなどをサポートする高級言語の特徴でヒープ言語と呼ばれる。他方PASCALやALGOL60では実行環境はスタックにとられFORTRANでは固定領域にとられる。(参1)

(4) タスク(task): 多重プログラミング又は多重プロセスングの環境において、計算機によって実行されるべき仕事の要素として制御プログラムによって取り扱われる命令の1つ以上の列(JIS)

(5) ガベージ(qarbage): どこからも参照されていないが、生成されてしまったオブジェクト。ガベージはガベージコレクタで回収する。(参1)

(6) ガベージコレクション(Garbage Collection): (オブジェクト指向システムに関連して)メモリ管理の中心となるプログラムを言う。主記憶を使いきるとメインの計算を止め、ガベージコレクタを動かして、もはや使われていないオブジェクトを全部集める方法である。この方法でガベージコレクタが働くと、計算が止まってしまうため入出力が全く反応しなくなり、実時間の応答が必要な用途には使えない。(参1)

(7) インタプリタ(interpreter): 解釈実行を行う翻訳プログラム(JIS)

(8) リアルタイム(real time): 計算機外部の他の処理との関係をもちながら、かつ外部の処理によって定められる時間要件に従って、計算機が行うデータの処理に関する用語。実時間。(JIS)

(9) オブジェクト(object): プロシージャ(定義した手続き)とデータの特性を結合させるエンティティ(実体)であり、これにより計算が行われ、局部的状態が蓄えられる。(参1)

(10) クラス(class): 同一の手続き群とデータ構造を持つオブジェクトの集合。(参3)

(11) ヒープ領域(heap area): プログラム実行時に必要に応じて使用されるような作業領域。(参2)

(12) スケジュールする(scheduling): ディスパッチされるべきジョブ又はタスクを選択すること。(JIS)

(13) イベント(event): ハードウェア/ソフトウェアが、自分自身の状態の変化を他のハードウェア/ソフトウェアに通知すること。一般にこの際の通知では、イベントの種類やハードウェア/ソフトウェアの状態を表す各種パラメータをメッセージとしてまとめ、相手に送信する。そしてイベントの通知を受けた側では、メッセージのパラメータなどから適切な処理を行う。(参A)

(14) イベントフラグ(event flag): タスクが1つまたは複数の事象の発生を待ち合わせるための機能とその事象を通知する機能とからなるタスク間同期通信機構。

(参3)

(15) セマフォ(semaphore): 複数のプロセスやタスク

を並列に処理するシステムで、各プロセス間、タスク間の同期やメッセージ制御、割込処理を行うための仕組み。(参3)

(16) 仮想マシン(仮想機械)(virtual machine): 複数の特定のプラットフォーム(OSやハードウェア)に組み込まれた、特定のプラットフォームに依存しないアプリケーションプログラムを実行する環境。同じ仮想マシンさえ提供されていれば、プラットフォームが変わっても同じアプリケーションプログラムが実行できる。

10 【0004】(17) Java VM(Java Virtual Machine): オペレーティングシステムに組み込まれた、Javaアプリケーションプログラムを実行するための環境。一般的なプログラムは、ソースコードをコンパイルして、それぞれのオペレーティングシステムに最適化した実行コードを生成する、というスタイルを採る。Javaで書かれたプログラムも同様の手順で作成するが、コンパイル後のコードは、特定のオペレーティングシステムに依存しない中間コードの形をとる。これをロードし、各オペレーティングシステムに合わせたコードに変換しながら実行するのがJava仮想マシンの役目であり、同じ仮想マシンさえ提供されていれば、プラットフォームが変わっても同じコードが実行できるようになっている。

【0005】(18) フリー領域(free area): ヒープ領域上の使用可能な領域、未使用領域。

【0006】(19) ノーマルスレッド(normal thread): リアルタイム性が要求されない処理を行うスレッド。

【0007】(20) マークテーブル(mark table): オブジェクトがどこからも参照されないか否かを調べるために存在するオブジェクトに1対1に対応する表。或るオブジェクトに参照があることが確かめられた場合、そのオブジェクトに対応する表の欄にマークを付ける。全ての参照関係を調べたときマークの無いオブジェクトは不要であるので取り除くことができる。

【0008】(21) オブジェクトの寿命(life time of the object): オブジェクトが生成され、消去されるまでの時間。

【0009】(22) ライトバリア(write barrier): オブジェクトへの参照関係の変更をチェックし、書き替えが起こった場合に何らかの処理をすること。本件の場合、書き替えが起きたとき、書き替える参照が指しているオブジェクトに対応するマークテーブルの欄にマークを付ける。

【0010】(23) スイープ(sweeping): ヒープ領域上の不要なオブジェクトを取り除く処理。

【0011】(24) オブジェクト生成(create the object): 新しいオブジェクトを生成すること。具体的には、ヒープ領域の一部をオブジェクトに割り当て、内容を初期化すること。

50 【0012】(25) オブジェクト消去(delete the object)

t) : 不要なオブジェクトを取り除くこと。具体的には、ヒープ領域に確保してある領域を解放すること。

【0013】(26) 参照(reference) : 或るオブジェクトAが別の特定のオブジェクトBにアクセスするためにオブジェクトBを特定する情報。具体的には、オブジェクトBを指すポインタまたはインデックス。

【0014】(27) 参照変更(change the reference / reconnect the reference) : 参照を現在のオブジェクトBから別のオブジェクトCに変更すること。

【0015】さて、従来のシングルプロセッサのコンピュータシステムにおいて、オペレーティングシステム上で複数のスレッドを並行処理する場合、共有メモリを用いて排他制御するために、また複数のタスク間で同期をとるために、従来よりセマフォやイベントフラグなどを用いて排他制御が行われている。

【0016】また、例えばプログラムを少量のメモリ環境下で動作させることなどを目的として、仮想記憶を行わないで単一のアドレス空間のメモリをプログラムの実行時に動的に割り当てる、いわゆる動的記憶管理が従来より行われている。このような動的記憶管理によれば、プログラムにより明示的にメモリ領域の解放を行わなければ、プログラムの実行過程において使用されなくなったメモリ領域が発生する。その結果、プログラムが使用できるフリー領域が次第に不足する。こうした問題を回避するために、使用されなくなったメモリ領域(ガベージ)を抽出し、これらを集めて(コレクトして)再び使用可能なフリー領域とする、ガベージコレクションと呼ばれる処理を自動的に行うようにしている。

【0017】ここで従来のガベージコレクションの処理手順をフローチャートとして図49に示す。ガベージコレクションのアルゴリズムにはマーク&スイープ法、複写法、参照カウント法などの各種方法が考えられているが、ここではマーク&スイープ法について例示する。図54に示すように、まずガベージコレクションの途中でガベージコレクションスレッド以外の他のスレッドが実行されないように割り込みを禁止し、シングルスレッドモードにする(s201)。続いてメモリ上のガベージコレクションの対象となる領域(以下「ヒープ領域」という。)に割り当てられているオブジェクトにそれぞれ対応するマークの記憶領域(以下「マークテーブル」という。)をクリアする(s202)。続いてヒープ領域内に割り当てられているオブジェクトの参照関係を示す情報を基に何れかのオブジェクトから参照されているオブジェクトを検出し、それらに対応するマークテーブル上の位置にマークを付ける処理を行う(s203)。この処理により、何れのオブジェクトからも参照されていないオブジェクトはもはや使われなくなったオブジェクトであり、それに相当するマークテーブルにはマークが付けられないことになる。従ってそのマークしていないオブジェクトを新たなオブジェクトの割当可能な領域、すなわちフリー領

域として抽出する(s204)。例えばこのフリー領域をリスト構造のデータとして生成する。その後、割り込み禁止を解除し、マルチスレッドのモードに戻す(s205)。

【0018】従来はこのようなガベージコレクションが、メモリのフリー領域が所定量まで減少した時点で自動的に起動されるようにしていた。

【0019】また、ガベージコレクションを行ったとき、任意の大きさ(メモリサイズ)のオブジェクトが任意に解放されるのでフラグメントが発生する。そこで、連続したサイズの大きな領域がとれるように、オブジェクトの割当領域を例えば先頭から順次詰めるメモリコンパクション(以下単に「コンパクション」という。)を実行していた。

【0020】

【発明が解決しようとする課題】上述した排他制御の機能をセマフォやイベントフラグなどのコンピュータ資源を用いることで実現している従来のシステムにおいては、その資源が他のスレッドなどで使用されている場合には、他のスレッドはその資源が解放されるのを待たなければならない。このような資源待ちの時間が生じると、リアルタイム性の要求されるシステムにおいては大きな障害となる。すなわち、資源待ち状態にあるスレッドは、その資源が解放されるまで、処理できずリアルタイムな応答が不可能となる。

【0021】例えば上記従来のガベージコレクション(以下「GC」という。)の方法によれば、メモリ空間が広いほど、ガベージとしてコレクトしてもよい領域を見つけ出すのに時間がかかり、例えば64~128MBのヒープ領域で数秒間を要し、且つGCはフリー領域がある程度減少した時点で不定期に行われるので、リアルタイム性の要求されるシステムには用いることができなかった。

【0022】リアルタイム性の要求されるシステムでは、あるタスクを実行中に何らかのイベント(割り込み)が発生すれば、そのイベントに応じた他のスレッドを処理することになるが、そのスレッドの切替に要する時間は、最悪値として例えば数十μsec以下であることが要求される。ところが、上述したようにGCが何時起動されるか予測できず、一旦起動されれば、CPUは数秒間GCに専念することになるため、その間リアルタイム処理は不可能となる。

【0023】上述の問題はGCの場合に限らず、長時間の資源待ち状態が生じるシステムに共通の問題である。

【0024】この発明の目的はセマフォやイベントフラグなどのコンピュータ資源をロックのメカニズムとして使用せず、処理の排他性を保証することにより、上述の問題を解消することにある。

【0025】GCの他の方法として、従来の複写法をインクリメンタルに行うようにして、リアルタイム性を確保する方法が情報処理Vol.35 No.11 pp1008 ~pp1010に

示されている。この方法によれば、GCの途中で中断を許すことができるので、時分割的に他のスレッドと並行処理することが一応はできる。しかし、この複写法をインクリメンタルに行うようにする方法では、複写中に他のスレッドがメモリを使用する訳にはいかないので、メモリを使用しない極一部のスレッドしか並行処理できない。また、複写法では、複写元と複写先のメモリ領域を確保しておかなければならないので、メモリの使用効率が低く、少量のメモリ環境下で動作させるシステムには向かない。

【0026】また、マーク&スイープ法でマルチスレッドに対応するために参照関係が変更される度にマークを付与する方法は、「On-the-fly GC」という名前で、GC専用のCPUに処理を割り当てるマルチCPU化のための方法が情報処理Vol.35 No.11 pp1006 ~pp1008に示されている。しかし、この方法では、オブジェクトが常に作成されて、且つ参照関係が変更されている場合に、既にマークしているツリーを何回も辿り直して新しいノードを見つけてマークしなければならず、マーク作業がいつまでも終わらない場合が生じたり、非常に長い時間がかかる、という問題があった。また、この方法ではスイープ中にシステムをロックする必要があった。

【0027】そこで、この発明の他の目的はマーク&スイープ法でありながらGCスレッドを他のスレッドと実質的に並行処理可能とし、GCの任意の時点で中断しても短時間に確実にGCを完了できるようにしたインクリメンタルなGCを可能とすることにある。

【0028】

【0029】

【0030】

【0031】

【0032】ここで、先行技術調査を行った際に発見した文献と本願発明との関係について示しておく。

【0033】(1) Incremental Garbage Collection of Concurrent Objects for Real-Time Application

この論文はBaker が書いたという1978年のリアルタイムGCに対して、全体の処理時間から必要なGCの処理時間の割合を求めるものである。本願発明に係る課題を解決するものではない。

【0034】(2) Distributed Garbage Collection for the Parallel Inference Machine:PIE64

GCを行う領域を細かく分けることにより、個々の処理時間を短くしてリアルタイム性を向上する方法である。本願発明のように全ての領域を対象にするものとは基本的に異なる。また、スケジューリングについては述べられていない。

【0035】(3) Garbage Collection in Distributed Environment

Baker の考えを発展させ、ネットワークの分散環境に適応したもの。ネットワーク固有の問題を解決しようとす

るものであり、本願発明とは異なる。この分散環境の考え方に、本願発明のスケジューラを組み合わせることにより、分散環境でより高度なリアルタイムCGを実現することも可能となる。

【0036】(4) 特開平1-220039号

システムコール発行時に、そのシステムコールにより起動されるタスクの優先度を制御するもの。優先度制御という点で共通点があるだけ。

【0037】(5) 特開平3-231333号

10 オブジェクトのサイズを検出し、そのサイズに応じてワークエリアをメモリに確保することが一応示されている。

【0038】(6) 電子情報通信学会誌Vol.80 No.6 pp586-592

マルチメディアオペレーティングシステムのコンセプトが示されている。

【0039】(7) 日本ソフトウェア科学会第12回D7-4

20 スタック上のオブジェクトの一番上にオブジェクトの大きさを書いておくことが示されている。すなわちオブジェクトのサイズを記憶しておく、という点でのみ関連がある。

【0040】(8) 11TH Real-Time System Symposium "Incremental Garbage Collection of Concurrent Object for Real-Time Applications"

オブジェクト間の参照ツリーに類似の考え方が示されている。

【0041】(9) 情報処理学会第38回全国大会5U-7

30 オブジェクトの寿命に関連してメモリ割当を行うことが示されている。

【0042】(10)Lecture Notes in Computer Science 259 "Garbage Collection in a Distributed Environment"

アプリケーションプログラムインタフェースの呼出に応じてGCを制御することと、オブジェクト参照に関する技術思想が示されている。

【0043】

【課題を解決するための手段】この発明の請求項1、4、5に係る発明は、或るスレッドからの、コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しに回答して、コンテキストスイッチの発生有無を示すフラグをコンテキストスイッチが発生していない状態に設定し、前記フラグがコンテキストスイッチの発生していない状態に設定された後、スケジューラによってコンテキストスイッチが行われたとき、前記フラグをコンテキストスイッチが発生した状態に設定するようにし、前記スレッドからの、コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しに回答して、前記フラグの状態に応じた値を前記スレ

ッドに返す。

【0044】これにより、コンピュータ資源をロックのメカニズムとして使用せずに、処理の排他性を保証する。すなわち或るスレッドAからのコンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェース（以下「API」という。）の呼び出しからコンテキストスイッチ発生有無検出の終了を依頼するAPIの呼び出しまでの間で行われたスレッドAの処理の途中で、コンテキストスイッチがあったか否かを、そのスレッドAで分かるようにする。もしコンテキストスイッチが発生しなければ、スレッドの切替は行われておらず、排他性が保たれている（例えば上記APIを発行したスレッドAが使用しているメモリ内容が他のスレッドBにより書き替えられていない）ことが分かる。逆に、コンテキストスイッチが発生していれば、その間のスレッドの処理を無効にして、例えばその処理を再度実行するなどの方法によって高い応答性を保ちながら、排他制御を行うことができるようにする。

【0045】また、或るスレッドの優先度を高い状態と低い状態とに交互に変更するようにしておき、コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しから前記コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しまでの予定されている処理時間を受け取って、前記スレッドの優先度が高い状態のとき、該スレッドの優先度が低い状態になるまでの残時間と前記処理時間とを比較し、前記残時間が前記処理時間より短いことを検知したときに、前記スレッドの優先度を低くする。

【0046】これにより、上記処理時間内に終了できない状態は、その処理が行われることがないため、その分のCPUパワーを無駄にすることがなく、全体の処理を進められる。

【0047】請求項2に係る発明は、メモリのヒープ領域内の他のいずれかのオブジェクトから参照されているオブジェクトを検出し、当該オブジェクトを前記ヒープ領域内の所定領域に複写する複写方式のガベージコレクションスレッドの際に、上記のコンテキストスイッチの検出による処理を行う。これにより、システムをロックすることなく、GCを開始することができ、リアルタイム性を保証することができる。

【0048】請求項3に係る発明は、メモリのヒープ領域内のどのオブジェクトからも参照されないオブジェクトのメモリ領域を他のオブジェクトのメモリ割り当て可能なフリー領域として解放することにより生じるフラグメントを解消するメモリコンパクションを行う際に、上記のコンテキストスイッチの検出による処理を行う。これにより、システムをロックすることなく、メモリコンパクションを開始することができ、リアルタイム性を保証することができる。

【0049】

【0050】

【0051】

【0052】

【0053】

【0054】

【0055】

【0056】

【0057】

10 【0058】

【0059】

【0060】

【0061】

【0062】

【0063】

【0064】

【0065】

【0066】

【0067】

20 【0068】

【0069】

【0070】

【0071】

【0072】

【0073】

【0074】

【0075】

【発明の実施の形態】この発明の実施形態であるプログラム制御装置およびメモリ割り当て装置の構成を図1～図5を参照して順次説明する。

【第1の実施形態】

【0076】図1は装置のハードウェアの構成を示すブロック図である。装置は基本的にCPU1とオブジェクト群を生成するヒープ領域およびマークテーブル等を記憶するメモリ2、および外部との入出力を行うI/O3とから構成する。また、外部から必要なプログラムをメモリにロードする場合は、CD-ROM読取インタフェース4を用い、プログラムが予め書き込まれたCD-ROM5を読み取るようにする。このCD-ROMが本願

40 発明に係るプログラム記録媒体に相当する。

【0077】図2はソフトウェアの構成を示すブロック図である。同図において、カーネル部分はCPUやメモリを資源として管理し、時分割によるマルチスレッドの機能を実現する。VM（仮想マシン）部分はプログラムとカーネルとのインタフェースを行うソフトウェアであり、ここではアプリケーションプログラムから見てVM以下の階層全体が例えばJava仮想マシンとして作

50 用される。このVMにはプログラムがバイトコード等の中間コ

ードで与えられるとき、これを解釈するインタプリタとその解釈に応じて呼び出されるプログラムモジュール等を含む。同図におけるプログラムは中間コードによる各種スレッドであり、上記インタプリタを介して、内部のプログラムモジュールを実行する。

【0078】図3はメモリのヒープ領域に生成されるオブジェクトの参照関係およびスタックとの関係を示す図である。ヒープ領域に対してオブジェクトを生成した際、或るオブジェクトから他のオブジェクトへの参照関係は同図に示すようにルートノードから延びるツリー構造を成す。例えばグローバル変数を宣言すれば、その変数に対応するオブジェクトが生成される。またスレッド毎に引き数領域、戻り番地、ローカル変数、作業領域等の情報を記憶するスタックが生成され、例えば図中矢印で示すようなスタック上のローカル変数からツリー上のグローバル変数への参照関係なども記憶される。これらのスタックはヒープ領域外の所定領域に格納される。

【0079】図4は図2に示したソフトウェアの構成をブロック図として詳細に示したものである。同図において、GCモジュール10はGCを行うための各種処理のプログラムモジュールであり、GCスレッド11はこれらのモジュールを呼び出すことによって、GCを実行させる。また、インタプリタ12は、この例で、「スレッド4」からオブジェクト生成依頼があったとき、GCモジュール10の「オブジェクト生成」を呼び出し、またオブジェクト間の参照関係の変更依頼があったとき、GCモジュール10の「マーク付与」を呼び出す。

【0080】なお、図4に示した例では、各スレッドが中間コード（例えばJavaアプレットの場合バイトコード）で表されているが、中間コードをVMに対するネイティブコードに変換するコンパイラを設けてもよい。（例えばJavaの場合、JIT(Just-In-Time コンパイラ)等を設ける。）この場合、ネイティブコードで記述したスレッドであるので、図4に示したインタプリタ12を介さずにGCモジュール10を直接アクセスすることになる。

【0081】図5はGCを行うことによって、ヒープ領域内に生じるフラグメントを解消するためのコンパクションを行った例を示す図である。図中ハッチング部分がオブジェクトであり、これをコンパクションすることによって、(B)に示すようにフラグメントが解消されて、連続したメモリ空間が広がることになる。

【0082】上記コンパクションは図4に示したGCモジュールの「コンパクション」のプログラムにより行う。

【0083】図6はコンテキストスイッチ発生有無を検出するためのAPIの使用例を示す図である。同図の(A)に示すように、処理1を実行する前にコンテキストスイッチ発生有無の検出開始を依頼するAPI #Aを発行してから処理1を実行する。処理1の終了後、コン

テキストスイッチ発生有無検出の終了を依頼するAPI #Bを発行する。(A)に示す例ではこの間にコンテキストスイッチが発生していないので、そのまま次の処理2を行う。もし、(B)に示すように、処理1の途中でスレッド#2の処理が行われれば、すなわちコンテキストスイッチが発生していれば、API #Bの発行後、処理1を破棄する。たとえばメモリの領域Aの内容を領域Bにコピーするような処理で、コピー処理中にコンテキストスイッチが発生した場合、領域Aの内容が変わって領域AとBの内容が不一致となる場合がある。不一致ではコピーしたことにならないので、領域Bを無効にする。このことは、最初から処理が行われなかったのと同じであり、処理自体を破棄したことに他ならない。

【0084】図7は上記の処理をフローチャートとして示したものである。まず、API #Aを発行してから処理1を実行する(s1→s2)。この処理1が終了した後、API #Bを発行して、その戻り値を取得する(s3)。戻り値がコンテキストスイッチの発生を表す場合、処理1を破棄して、再び処理1を実行する(s4→s5→s1→s2)。戻り値がコンテキストスイッチの非発生を表す場合、処理を終了する。このように所定の期間内でのコンテキストスイッチの発生有無が分かるので、コンテキストスイッチが発生すれば、その間の処理を破棄し無効とすることによって、システムを現実にはロックしていないにも拘らず排他的制御を行うことが可能となる。

【0085】図8は上記API #AおよびAPI #Bのカーネルにおける処理手順を示すフローチャートである。API #Aの発行(システムコール)があれば、コンテキストスイッチの発生有無を示すフラグをクリアする。また、API #Bが発行されると、上記フラグの状態を戻り値としてスレッドに返す。

【0086】図9はコンテキストスイッチの処理手順を示すフローチャートである。コンテキストスイッチがスケジューラによって行われると、上記フラグをセットしてからコンテキストスイッチを実行する。すなわちスイッチ前のスレッドの実行状態をコンテキストとして格納するとともに、スイッチ後のスレッドのコンテキストを読み出してCPUのレジスタ等に設定する。

【0087】図10は上記コンパクションの処理手順を示すフローチャートである。まず、ヒープ領域内の先頭のオブジェクトを指定し(s11)、そのオブジェクトをヒープ領域の先頭にコピーするための領域を確保し(s12)、コピー中に他のスレッドによってその領域に何らかのデータが書き込まれないようにしてから上記API #Aを発行する(s13)。そして、図5に示したように、ヒープ領域内のオブジェクトを先頭から順次空き領域へコピーすることによって詰めていく(s14)。1つのオブジェクトについてのコピーが終われば、上記API #Bを発行する(s15)。このことに



より、API # Aを発行してから、API # Bを発行するまでの期間にコンテキストスイッチが発生したか否かがAPI # Bの戻り値として得られる。もしコンテキストスイッチが発生していれば、今回コピーを行ったオブジェクトを既に確保している領域に再びコピーする(s 16→s 13→...)。もしコンテキストスイッチが発生していなければ、次のオブジェクトについて同様に処理を行う(s 16→s 17→s 18→...)。このようにシステムをロックすることなく、しかも他のスレッドとともに、コンパクションを同時に行うことが可能となる。

【0088】上述の例ではマーク&スイープ法によるGCにおけるコンパクションに適用したが、複写法によるGCに適用する場合、図11および図12に示す処理を行う。

【0089】図11は上記複写法によるGCのフローチャートである。まず、オブジェクトの参照関係を表すツリー構造のデータのルートノードへポインタを移動し(s 21)、そのルートノードに相当する、From領域にあるオブジェクトをTo領域に複写する(s 22)。(ヒープ領域はFrom領域とTo領域に分けられ、From領域内の残すべきオブジェクトのみをTo領域に複写することによってTo領域にガベージのないオブジェクトだけを再構成する。次回は現在のFrom領域をTo領域とし、To領域をFrom領域として、その操作を交互に繰り返すのが、複写法によるガベージコレクションである。)その後、ツリーを辿って、参照関係にある次のオブジェクトへポインタを移動させ(s 23)、そのオブジェクトをTo領域に複写する(s 24)。この処理をツリーで辿れるすべてのオブジェクトについて行う。

【0090】図12は図11における複写処理の内容を示すフローチャートである。まず、複写すべきオブジェクトをTo領域内の所定位置にコピーするための領域を確保し、コピー中に他のスレッドによってその領域に何らかのデータが書き込まれないようにし、上記API # Aを発行してから(s 31)、オブジェクトをFrom領域からTo領域に複写する(s 32)。その後、上記API # Bを発行する(s 33)。このことにより、API # Aを発行してから、API # Bを発行するまでの期間にコンテキストスイッチが発生したか否かがAPI # Bの戻り値として得られる。もしコンテキストスイッチが発生していれば、今回複写を行ったオブジェクトを既に確保している領域に再び複写する(s 34→s 31→...)。

【0091】図13および図14はコンテキストスイッチの発生有無を検出して排他性を確保するのではなく、複写しようとするオブジェクトが複写前に書き替えられたか否かを検出して排他性を確保する場合の処理手順を示すフローチャートである。

【0092】複写を行う場合、先ず複写すべきオブジェクトをTo領域内の所定位置にコピーするための領域を確保し、コピー中に他のスレッドによってその領域に何らかのデータが書き込まれないようにしてから、図13に示すようにAPI # Cを発行する(s 41)。このAPI # Cは指定したメモリ領域ない対する書き込みが発生したか否かの検出を依頼するアプリケーションプログラムインタフェースである。続いてオブジェクトをFrom領域からTo領域に複写する(s 42)。その後、上記API # Dを発行する(s 43)。このAPI # Dは上記API # Cが呼び出されてから、このAPI # Dが呼び出されるまでの間に指定メモリ領域に何らかのデータの書き込みが発生したか否かが戻り値として返されるアプリケーションプログラムインタフェースである。したがって、複写しようとするオブジェクトのメモリ領域を指定してAPI # Cを発行し、API # Dも戻り値を見れば、そのオブジェクトが参照されたか否かが判る。オブジェクトが参照されていれば、内容が書き変わっている可能性があるため、そのオブジェクトの複写をやり直す(s 44→s 41→...)。

【0093】図14は上記API # CおよびAPI # Dのカーネルにおける処理手順を示すフローチャートである。API # Cの発行(システムコール)があれば、所定にワークエリアをクリアし、API # Cのパラメータで指定された領域がライトされたときに例外が発生するようにMMU (Memory Management Unit) を設定する。また、API # Dが発行されると、上記パラメータで指定された領域がライトされたときに例外が発生しないように、MMUに対する上記設定を解除する。そして上記ワーク領域にセットされるフラグの状態を戻り値としてスレッドに返す。

【0094】図15は上記例外発生時のMMUの処理内容を示すフローチャートである。例外が発生すると、上記ワーク領域内の参照フラグをセットする。

【0095】なお、上述の例では複写法によるGCの場合について示したが、マーク&スイープ法におけるコンパクションの場合にも同様に適用できる。

【0096】〔第2の実施形態〕図16および図17はインクリメンタルにガベージコレクションできるGCスレッドの優先度を自動的に切り替えるようにした場合の例を示す図である。

【0097】図16の(A)に示すように、GCスレッドの優先度を、高優先度時間は高め(例えば最高優先度にし)、低優先度時間は低く(例えば最低優先度)する動作を交互に行う。

【0098】同図の(B)はGCスレッド以外の中程度の優先度のスレッドを同時に行った場合の例について示している。すなわちGCスレッドの優先度が低いとき、それより優先度の高いスレッドがReady状態となれば、コンテキストがスイッチされ、そのスレッドの実行

中にGCスレッドの優先度が高くなれば、そのGCスレッドにコンテキストがスイッチされ、そのGCスレッドの処理が中断すると、上記のGCスレッド以外のスレッドの処理を行うことになる。このようにすれば、一定周期で必ずGCスレッドが実行されるため、フリー領域が慢性的に不足状態となることが防止され、常に高いパフォーマンスを維持することができる。

【0099】図17はスレッドの優先度の切替に関するカーネルの行う処理手順を示すフローチャートである。優先度の値は複数段階あり、その値の設定は対応するAPIの発行により行えるようにしている。ここでは、GCスレッドの2つの優先度を設定する際、高優先度の値を設定するAPIが発行されると、図17の(A)に示すように、その値をGCスレッドの高優先度の値として設定する。同様に、低優先度の値を設定するAPIが発行されると、(B)に示すように、その値をGCスレッドの低優先度の値として設定する。また、GCスレッドの高優先度の時間を設定するAPIが発行されると、同図の(C)に示すように、その値を設定し、同様に低優先度の時間を設定するAPIが発行されると、同図の(D)に示すように、その値を設定する。

【0100】図17の(E)は、カーネルのスケジューラに対する処理手順を示すフローチャートである。ここでは、初期状態で、高優先度のキューにGCスレッドが資源の割当を受けようとする待ち行列に入っているものとし、まず、そのデータ(GCスレッドを識別するデータ)を高優先度のキューから取り出して低優先度のキューに挿入する(s51)。そしてスケジューラは、既に設定されている低優先度時間だけ時間待ちを行い(s52)、続いて低優先度のキューからGCスレッドを識別するデータを取り出して高優先度のキューに挿入する(s53)。そしてスケジューラは、既に設定されている高優先度時間だけ時間待ちを行う(s54)。以上の処理を繰り返すことによって、スケジューラの動作により図16(A)に示したようにGCスレッドの優先度が交互に切り替わる。GCスレッド以外のスレッドについては、そのスレッドの優先度に対応するキューの待ち行列に応じて従来と同様のスケジューリングが行われ、図16の(B)に示したように、コンテキストスイッチがなされる。

【0101】図18は上記のGCスレッドの高優先度時間をAPIによって設定可能とした場合について示している。また、図19はそのAPIの呼び出しに応じたカーネルにおける処理手順を示すフローチャートである。このAPIの発行(システムコール)があれば、上記の高優先度時間をそのAPIのパラメータによって設定(登録)する。

【0102】図20は上記のGCスレッドの高優先度時間を設定可能とするAPIを使うプログラムの例を示すフローチャートである。まず、ループの1回目の示すフ

ラグの状態を見る(s61)。最初はOFF状態であるので、このフラグをONし(s62)、現在のフリー領域を調べ、それを記録する(s63)。初めはGCの高優先度時間と低優先度時間は予め定められたデフォルト値である。スケジューラにより、一定時間停止した後(s64)、今度は、前回に調べたフリー領域の容量と現在のフリー領域の容量との大小比較を行い(s65)、フリー領域の容量が増加していれば、GCスレッドの高優先度時間を短くし(s66→s67)、フリー領域の容量が減少していれば、GCスレッドの高優先度時間を長くする(s68)。以上の処理を繰り返す。このことによって、GCの優先度が動的に自動調整される。

【0103】図21はGCスレッドの高優先度時間と低優先度時間とによる周期を設定可能とした場合について示している。(A)は周期が長い場合、(B)は短い場合である。

【0104】図22は上記周期を設定可能とする周期設定APIの呼び出しに応じたカーネルにおける処理手順を示すフローチャートである。この周期設定APIの発行があれば、現在設定されている高優先度時間および低優先度時間の値を読み取り、その割合(比率)を計算する(s71→s72→s73)。その後、この周期設定APIのパラメータで指定された周期の値に応じて高・低優先度時間を再計算する(s74)。そして、その高優先度時間および低優先度時間を設定更新する(s75→s76)。

【0105】たとえば連続的なパルス処理するような、連続してCPU資源が必要なアプリケーションプログラムの場合には、GCスレッドの周期が短くなるように周期設定APIを発行し、通常のアプリケーションプログラムのように断続的にCPU資源を利用するアプリケーションプログラムでは周期が長くなるように周期設定APIを発行する。

【0106】〔第3の実施形態〕図23および図24は必要な時点でGCスレッドを実行し、且つリアルタイム性を確保するようにした例であり、図23に示す例では、スレッド1とスレッド3がリアルタイム性の要求されるスレッド、スレッド2がリアルタイム性の要求されないスレッド(ノーマルスレッド)である。また、スレッド3がGCスレッドである。メモリのフリー領域が多い通常状態で、スレッド2が実行されているときにイベント1が発生すれば処理がスレッド1に移され、イベント1の処理に起因するスレッド1の処理が終了すればスレッド2へ戻る。同様に、イベント2が発生すればスレッド3に処理が移る。もしスレッド2の処理によってフリー領域が予め定めた警告レベルまで低下したとき、スレッド2の処理を中断して、スレッド3のGCスレッドを実行する。この処理によってフリー領域が確保されるとスレッド2の処理へ戻る。もしGCスレッドの処理途

中でイベント1が発生すれば、GCの途中でであってもスレッド1へ処理が移る。このようにリアルタイム性の要求されるスレッドでは、一般に生成されるオブジェクトの量が少なく、大体の予測が可能であるので、それに応じて上記警告レベルを設定しておけば、スレッド2に処理によって、フリー領域が大幅に減少してスレッド1やスレッド3のリアルタイム性の要求される処理が実行できなくなるといった、不都合が回避できる。

【0107】図24は上記コンテキストスイッチの処理を行うスケジューラの処理手順を示すフローチャートである。まずイベントの発生有無を検知し(s81)、イベントが発生していれば、対応するリアルタイムスレッドにコンテキストをスイッチする(s82)。イベントが発生していなくて、フリー領域が警告レベルより多ければ、ノーマルスレッドのうち優先度の最も高いもの(図23に示した例ではスレッド2)にコンテキストをスイッチする(s83→s84)。もしフリー領域が警告レベルより少なくなれば、GCスレッドにコンテキストをスイッチする(s85)。

【0108】〔第4の実施形態〕図25はコンテキストスイッチの検出を依頼するAPIにおける強制コンテキストスイッチの例を示している。先に示した例では、API#Aを発行してから、API#Bを発行するまでの間にコンテキストスイッチが発生したか否かが判るようにしたものであったが、この図25に示す例は、GCの優先度を高・低交互に切り替える場合に、上記の排他制御のためのAPIを発行している間にコンテキストスイッチが発生することを予測して、無駄な処理を行わないようにしたものである。すなわち、高優先度GCスレッドを実行中にAPI#A'を発行した際、API#A'は高優先度時間が終了するまでに、必要な処理が完了するか否かを判定して、もし完了しなければ、その時点でGCスレッドの優先度を低くする。図25に示した例では、GCスレッドの優先度が低くなったことにより、GCスレッド以外の中優先度のスレッドにコンテキストスイッチされることになる。このことにより無駄な処理が避けられる。

【0109】図26は上記API#A'の呼び出しに応じたカーネルにおける処理手順を示すフローチャートである。このAPIの呼び出しがあれば、コンテキストスイッチフラグをクリアし(s91)、GCスレッドの高優先度時間の残時間を取得する(s92)。そして、このAPI#A'のパラメータである排他時間(API#A')を発行してから、API#Bを発行するまでの時間と上記残時間との大小比較を行う(s93)。もし、残時間が排他時間より短ければGCスレッドの優先度を強制的に低くする(s94)。なお、API#B呼び出しによる処理内容とコンテキストスイッチの処理内容は図8および図9に示したものと同様である。

【0110】〔第5の実施形態〕図27はメモリ(ヒー

ブ領域)使用量に応じてGCのアルゴリズムを切り替えるための処理手順を示すフローチャートである。まず、メモリ使用量の値とそれに応じてどのアルゴリズムでGCを行うかを示すしきい値を設定し(s101)、メモリ使用量を取得する(s102)。これはヒープ領域内に生成されたオブジェクトのメモリサイズの合計値である。メモリ使用量がしきい値th1を超えた場合、GCアルゴリズム(1)の手順でGCを行う(s103→s104)。メモリ使用量がしきい値th2を超えたなら、GCアルゴリズム(2)の手順でGCを行う(s105→s106)。以下同様である。ここで、しきい値th1はしきい値th2より小さく、GCアルゴリズム(1)としては図16に示した、GCスレッドの優先度の高低を交互に繰り返す処理を行う。しきい値の高いときに行うGCアルゴリズムとしては、図23に示した不定期のGCを実行する。ここでGC自体は例えばマーク&スイープ法により行う。通常、前者の場合にはCPU負荷が軽く、専らこの方法によりGCが行われるが、ノーマルスレッドによって短時間に大量のメモリが使用された場合には、図23に示した方法によりGCを重点的に行う。このことによって常に広いフリー領域を確保し、且つリアルタイム性を維持する。

【0111】〔第6の実施形態〕図28～図30はオブジェクト生成時のヒープ領域に対する新たなオブジェクトの割り当てサイズを定めるための処理に関する図である。一般にプログラムの実行により生成されるオブジェクトのサイズの分布は図28に示すように略正規分布を示す。その中心値は32と64バイトの間に納まる程度である。そこで、図29の(A)に示すように、この中心値より大きなサイズで、且つ2の巾乗バイトの整数倍のサイズをオブジェクトの割り当てサイズとする。従来は同図の(B)に示すように、生成すべきオブジェクトのサイズの量だけ任意に割り当てられていたため、そのオブジェクトの消去の際、不揃いのサイズのフラグメントが生じる。本願発明によれば、新たに生成されるオブジェクトのサイズが上記固定値の整数倍であるため、メモリ領域の再利用性が高まり、メモリ使用効率が全体に高まる。しかも場合によってはコンパクションが不要となる。

【0112】図30はそのオブジェクト生成時の処理手順を示すフローチャートである。まず、これまでに生成したオブジェクトのサイズの発生頻度の分布データを求める(s111)。既に前回までに分布データを求めている場合には、それを更新する。続いて、今回割り当てべきメモリの空いている領域で、且つ生成しようとするオブジェクトのサイズより大きな領域を探し、上記の固定サイズの整数倍のメモリ領域にオブジェクトを割り当てる(s112→s113→s114→s115)。上記2の巾乗バイトはシステム定数であるが、必ずしもこの値を固定サイズとする必要はなく、任意である。固

定サイズをもし大き過ぎる値に採れば、サイズの大きな領域に小さなオブジェクトが割り当てられる場合が増えることになり、逆に、固定サイズを小さ過ぎる値に採れば、オブジェクトの生成に再利用できない領域が増えることになる。分布データを基に上記固定サイズを決定する場合には、メモリの使用効率が最適となるように決定すればよい。また、最適値でなくても、例えば2の巾乗の値を採れば、アドレス決定がやり易くなるという効果を奏する。

【0113】図31は上記オブジェクトのサイズの発生頻度分布データを求めるプログラムモジュール、およびオブジェクトの割り当てサイズを決定するプログラムモジュールの処理内容を示すフローチャートである。先ず、オブジェクトのサイズの発生頻度分布データを求めるプログラムモジュールをロードし(s121)、そのプログラムモジュールを起動する。すなわち一定時間が経過するまで(たとえばアプリケーションプログラムの起動・終了が10回行われるまで、またたとえば24時間経過するまで)、生成された各オブジェクトのサイズをサイズ毎に計数し、その分布データを求める(s122→s123)。その後、その分布データをシステムに登録し(s124)、オブジェクトのサイズの発生頻度分布データを求めるプログラムモジュールをアンロードする(s125)。続いて固定サイズを決定するプログラムモジュールをロードし(s126)、そのプログラムモジュールを実行する。すなわちシステムの登録された分布データを取得し、その分布中心値より大きなサイズで、且つたとえば2の巾乗バイトを固定サイズとして決定し、その値をシステムに登録する(s127→s128)。その後、固定サイズを決定するプログラムモジュールをアンロードする(s129)。

【0114】このように、一度オブジェクトのサイズの分布が検知された後、および固定サイズが決定された後は、それらのためのプログラムモジュールをアンロードすることによって、メモリ領域とCPUパワーの無駄を無くす。

【0115】〔第7の実施形態〕図49は上記固定サイズをAPIによって設定する例を示している。図に示すように、固定サイズ設定処理では、固定サイズを引数として固定サイズ設定APIを呼び出す。呼び出されたAPIでは、引数を固定サイズとしてシステムに登録する。オブジェクト生成時にはオブジェクトのサイズと固定サイズを比較し(s211)、固定サイズ以下であれば、ヒープ上の固定サイズの空き領域を探し、見つかった領域をオブジェクトに割り当てる(s212→s213)。固定サイズを超える場合は、ヒープ上のオブジェクトサイズより大きな空き領域を探し、見つかった領域をオブジェクトに割り当てる(s214→s215)。

【0116】〔第8の実施形態〕図50は上記固定サイズをAPIによって設定する他の例を示している。図5

0に示すように、この例では、まずオブジェクトサイズの分布データをオブジェクトサイズ分布設定APIを呼び出して設定する。この分布データは予め所定時間所定のアプリケーションを実行して測定したものである。オブジェクトサイズ分布設定APIは呼び出しに応答して、引数をオブジェクトサイズ配列変数にセットする。続いて、その中心値より大きなサイズで、且つ2の巾乗バイトを固定サイズとして決定し、これを固定サイズ変数にセットする。

【0117】〔第9の実施形態〕図51～図53はオブジェクトの割り当てサイズを予めいくつかのサイズに定めておく例を示す図である。図51は事前に所定のアプリケーションを所定時間実行して測定したオブジェクトサイズの分布および分割領域の集合を表している。予めヒープ領域を分割する場合、実際のオブジェクトサイズの分布を測定し、その分布に近くなるように、分割サイズと数を定める。たとえば、ヒープ領域が2MBの場合、分割サイズ指定変数を

集合No. n	バイトk	個数m
1	64	5000
2	256	10000
3	1k	10000
4	4k	5000
5	32k	500

とする。

【0118】上記分割サイズを設定する場合、図52に示すように、分割サイズ設定APIを呼び出して行う。呼び出された分割サイズ設定APIは図53に示すように、引数を分割サイズ指定変数にセットし、それに応じてヒープ領域を分割する。すなわち、まずループカウンタnを0とし(s221)、n番目の分割サイズ指定変数からサイズkと個数mを取得する(s222)。続いてヒープ領域からサイズkの領域をm個に分割し(m個分確保し)、リストに登録する(s223)。この処理をループカウンタnを1インクリメントしながら繰り返し、全てのサイズの分割を行う(s225→s222→...)。なお、上記の例で、サイズが32kバイトを超えるオブジェクトを生成する場合は、ヒープ領域内の上記分割された領域以外の領域に割り当てる。

【0119】〔第10の実施形態〕図32～図34はオブジェクトの寿命を考慮してGCの効率を高めるための処理を行う例を示す図である。図32の(A)に示す例では、ヒープ領域として、短寿命のオブジェクトを生成する領域と長寿命のオブジェクトを生成する領域とに分け、クラスは長寿命領域に確保する。尚、クラスはその他の固定領域に確保してもよい。そして、クラスにはオブジェクトを生成するためのテンプレートとしての定義情報以外に、生成するオブジェクトの寿命を示す寿命フラグを持たせる。この寿命フラグはクラスの生成時に自

動的に生成する。同図の(B)は従来のヒープ領域に対するオブジェクトの割り当て例を示す図である。

【0120】図33はオブジェクトの消去の手順を示すフローチャートである。同図に示すようにオブジェクトを消去した際に、そのオブジェクトの寿命が長いか短いかを判定し、寿命が短ければ、そのオブジェクトのクラスの寿命フラグを短寿命にセットする。例えばオブジェクトの領域内にそのオブジェクトが生成された時刻を格納しておき、そのオブジェクトを消去する時刻との差によって、そのオブジェクトの寿命を求める。上記時刻は例えばGCの回数を単位としてもよい。

【0121】図34はオブジェクトの生成の処理手順を示すフローチャートである。クラスの寿命フラグが短寿命を示していれば、短寿命領域にオブジェクトを生成し、そうでなければ、長寿命領域にオブジェクトを生成する。

【0122】このようにオブジェクトの寿命に応じてメモリの割り当て領域を区分することによって、長寿命領域ではフラグメントを大幅に低下でき、メモリの使用効率が向上する。また、例えば寿命領域についてGCを重点的に行うことにより、GCに消費されるCPUパワーを小さくすることができる。

【0123】【第11の実施形態】次に、マーク&スイープ法によるインクリメンタルGCについて図35~48を参照して説明する。

【0124】図35はマーク&スイープ法によるGCの全体の処理手順を示すフローチャートである。このGCはマークテーブルのクリア、上記ツリー探索によるマーク付与およびオブジェクトの消去(スイープ)を繰り返して行う。

【0125】図36は図35における「マーククリア」の処理内容を示すフローチャートである。この処理はマークテーブルの内容を一旦クリアするものであり、まずマークテーブルの先頭にポインタを移動し(s131)、その位置のマークをクリアし(s132)、次のマークの位置までポインタを移動させる(s133)。この処理を全てのマークについて繰り返す(s134→s132→...)。

【0126】図37は図35における「オブジェクトの消去」の処理内容を示すフローチャートである。まず、マークテーブルの先頭にポインタを移動させ(s141)、マークの有無を検出し、マークされていない場合は、そのマークテーブル上の位置に相当するオブジェクトのヒープ領域内の位置を計算し、該当のオブジェクトを消去する(s142→s143→s144)。続いて、マークテーブルのポインタを次に移動して、同様の処理を繰り返す(s145→s146→s142→...)。これによりマークテーブルにマークされているオブジェクトを残し、その他のオブジェクトをヒープ領域から消去する。

【0127】説明を容易にするために、先ず前提となるマーク&スイープ法におけるマーク付与について説明する。

【0128】図38はツリー探索によるマーク付与の手順を示す図である。(A)に示すように、ルートノード10から各ノードへ、ツリー構造で表される参照関係を辿って、参照関係にあるノード(オブジェクト)にマークを付与する。具体的にはマークテーブル上の該当位置のビットをセットする。このツリー構造は、たとえば或るオブジェクトが他のどのオブジェクトを参照しているかを示す、オブジェクト内に設けられる変数の内容によって構成され、このオブジェクトの参照関係を辿ることが、すなわちツリーを辿ることである。

【0129】(A)のように、ノード番号3までマーク付与を行った時点で割込が発生し、その割込処理によって、(B)のように、ルートノードからノード番号7で表されるオブジェクトへの参照関係が絶たれて、ノード番号2で表されるオブジェクトからノード番号7で表されるオブジェクトが参照される関係となれば、割込処理が終了してGCスレッドに戻って、マーク付与を再開したとき、ルートノードからノード番号7で表されるオブジェクトへの参照関係が絶たれているので、(C)に示すようにポインタをルートノードに戻して次の参照関係にあるノード番号8に進むことになる。この時点ではノード番号5、6に対してはマーク付与されない。そこで、参照関係の変更のあったオブジェクトについては、そのオブジェクトからツリーを辿ってそのオブジェクトから参照されているオブジェクトについてマークを付与する必要がある。

【0130】図39は「オブジェクトの生成」の処理内容を示すフローチャートである。まずカーネルに対してシステムのロックを起動し(s151)、ヒープ領域内の空いている領域を探し(s152)、生成しようとするオブジェクトのサイズより大きな領域に対して必要なサイズを割り当て(s153→s154)、参照変更のマーク付与(ライトバリア)を行い(s155)、システムをアンロックする(s156)。

【0131】図40は上記「参照変更のマーク付与」の処理手順を示すフローチャートである。先ず、参照変更されたオブジェクトからマークテーブル上の位置を計算し、該当のマークがWhiteであるか否かを判定する。このWhiteマークはたとえば00の2ビットで表され、未だマークされていない状態を意味する。もし、Whiteマークでなければ、すでにマークされているので、そのまま処理を終了する。Whiteマークであれば、それをGrayにマークする。このGrayマークはたとえば01の2ビットで表され、参照変更のあったオブジェクトであることを意味する。なお、オブジェクトからマークの位置を計算する際、たとえばオブジェクトのアドレスを1/8してオフセットを加えることによって行うか、オブジェクトの

通し番号により計算する。

【0132】図41は「ツリー探索によるマーク付与」の処理手順を示すフローチャートである。まず、ツリーを辿るためのポインタをツリーのルートノードに移動させ（s161）、新たに生成するオブジェクトに対応するマークを付与する（s162）。続いてツリーを辿って、ポインタを次のオブジェクトへ移動させ（s163）、この処理をツリーの最後まで繰り返す（s164→s162→・・・）。その後、各スレッドスタックの先頭へポインタを移動させ（s165）、スタックにあるオブジェクトに対応するマークを付与する（s166）。その後、ポインタをスタックの次に移動させて（s167）、この処理をツリーの最後まで繰り返す（s168→s166→・・・）。その後、スレッドスタックの次に移り（s169）、同様の処理をスレッドスタックの最後まで繰り返す（s170→s166→・・・）。さらにこのスレッドスタックについての処理をすべてのスレッドスタックについて行う（s171→s172→s165→・・・）。この一連のツリー探索において、Grayマークが途中で1度でも検出された場合、もう一度ルートノードから探索およびマーク付与を行う（s173→s161→・・・）。

【0133】図42は図41における「オブジェクトに対応するマーク付与」の処理手順を示すフローチャートである。この処理は、生成されているオブジェクトからマークテーブル上の位置を計算し、Black マークを付与する。このBlack マークはたとえば1xの2ビットで表され、マークされている状態を意味する。

【0134】上述したようなマーク付与の方法では、その途中で割込がかかってオブジェクトの参照関係が変化すると、Grayマークが付与されるので、ツリー探索を繰り返さなければならない。場合によってはいつまでもたってもマーク付与が完了せずに、GCがいつまでも行われなという事態に陥る。

【0135】図43は上記の問題を解消するための「ツリー探索によるマーク付与」の手順を示す図である。

(A)は図38に示した(A)の状態では割込がかかって参照関係が変化したときの状態を示す。このように、ノード番号3までマーク付与を行った時点で割込が発生し、その割込処理によって、ルートノードからノード番号7で表されるオブジェクトへの参照関係が絶たれて、ノード番号2で表されるオブジェクトからノード番号7で表されるオブジェクトが参照される関係となれば、参照先のノード番号7表すデータをマークスタックに積む。その後、割込処理が終了してGCスレッドに戻って、マーク付与を再開したとき、ルートノードからノード番号7で表されるオブジェクトへの参照関係が絶たれているので、(B)に示すようにポインタをルートノードに戻して次の参照関係にあるノード番号8をマークする。この時点で一通りのツリー探索を終了し、その後

は、(C)に示すようにマークスタックの内容によって示されるノードからツリーを辿って参照関係にあるオブジェクトについてマークを付与する。これにより(D)に示すように、参照関係にある全てのオブジェクトについてのマーク付与が完了する。

【0136】図44は「参照変更のマーク付与」の処理手順を示すフローチャートである。このように、参照変更されたオブジェクトを示すデータを上記マークスタックに積む。なお、オブジェクトの生成の処理は図39に示したもの変わらない。

【0137】図45は、「ツリー探索によるマーク付与」の処理手順を示すフローチャートである。ステップs161～s172の部分は図41に示したフローチャートのステップs161～s172と同じである。この図45に示す例では、全スレッドスタックについてのツリー探索を完了した後、マークスタックからデータを取り出し（s181→s182）、そのデータで示されるオブジェクトに対応するマーク付与を行い（s183）、そのオブジェクトからツリーを辿ってツリーの最後まで、参照関係にあるオブジェクトのマーク付与を行う（s184→s185→s183→・・・）。この処理をマークスタックが空になるまでマークスタックのポインタを更新しながら繰り返す（s181→s182→・・・）。

【0138】図46は図45における「オブジェクトに対応するマーク付与」の処理手順を示すフローチャートである。この処理は、生成されているオブジェクトからマークテーブル上の位置を計算し、マークを付与する。

【0139】このようにマークスタックを用いることによって、ツリー探索をルートノードから再開する必要がなくなり、マーク付与に要する全体の処理時間を大幅に短縮化できる。

【0140】図47および図48は上記マークスタックを用いてマーク付与を行う場合に、更に無駄な処理時間を無くすようするフローチャートである。図47は上記「参照変更のマーク付与」の処理手順を示すフローチャートである。先ず、参照変更されたオブジェクトからマークテーブル上の位置を計算し、該当のマークがWhiteであるか否かを判定する（s191→s192）。このWhite マークは上述したように未だマークされていない状態を意味する。もし、White マークでなければ、すでにマークされているので、そのまま処理を終了する。White マークであれば、それをGrayにマークする（s193）。上述したようにこのGrayマークは参照変更のあったオブジェクトであることを意味する。続いてマークスタックに参照先のオブジェクトを示すデータを積む（s194）。

【0141】図48は上記「オブジェクトに対応するマーク付与」の処理手順を示すフローチャートである。この処理は、生成されているオブジェクトからマークテ



ブル上の位置を計算し、Black マークを付与する。この Black マークは上述したように、マークされている状態を意味する。なお、「オブジェクトの生成」の処理は図 39 に示したものと同様である。

【0142】このように、参照変更のあったオブジェクトについてマークを付与する場合、そのオブジェクトが初めて検出されたオブジェクトである場合にのみマークを付与するようにしたため、マークスタックの内容によるツリー探索に要する時間およびマークスタックの読み出しに要する時間を短縮化できる。

【0143】なお、参照変更のあったオブジェクトについてのマークを記憶するのはスタックでなくてもよく、FIFO のバッファを用いてもよい。

【0144】また、実施形態ではマークテーブルにマークを付与するようにしたが、オブジェクトの内部にマーク用の情報を設けて、オブジェクトに直接マークを付与するようにしてもよい。

【0145】

【発明の効果】請求項 1, 4, 5 に係る発明によれば、或るスレッドからのコンテキストスイッチ発生有無検出の開始を依頼する API の呼び出しからコンテキストスイッチ発生有無検出の終了を依頼する API の呼び出しまでの間で行われたスレッドの処理の途中で、コンテキストスイッチがあったか否かが、そのスレッドで分かるため、コンピュータ資源をロックのメカニズムとして使用せずに、処理の排他性が保証される。

【0146】また、コンテキストスイッチが発生していれば、その間のスレッドの処理を無効にして、例えばその処理を再度実行するなどの方法によって高い応答性を保ちながら、排他制御を行うことができるようになる。

【0147】さらに、或るスレッドの優先度を高い状態と低い状態とに交互に変更するようにしておき、スレッドの優先度が高い状態のとき、該スレッドの優先度が低い状態になるまでの残時間が、必要な処理時間より短いとき、その処理が行われることがないため、その分の CPU パワーを無駄にすることがなく、全体の処理を効率よく進められる。

【0148】請求項 2 に係る発明によれば、システムをロックすることなく、GC を開始することができるので、リアルタイム性を保証することができる。

【0149】請求項 3 に係る発明によれば、システムをロックすることなく、メモリコンパクションを開始することができるので、リアルタイム性を保証することができる。

【0150】

【0151】

【0152】

【0153】

【0154】

【0155】

【0156】

【0157】

【0158】

【0159】

【0160】

【0161】

【0162】

【0163】

【図面の簡単な説明】

10 【図 1】実施形態に係る装置のハードウェアの構成を示すブロック図

【図 2】同装置のソフトウェアの構成を示すブロック図

【図 3】オブジェクト間の参照関係を示すツリーおよび各スレッドのスタックとの関係を示す図

【図 4】ソフトウェアの機能ブロック図

【図 5】コンパクションの作用を説明するための図

【図 6】コンテキストスイッチの有無によるスレッドの処理内容の変化の例を示す図

【図 7】排他制御の処理手順を示すフローチャート

20 【図 8】コンテキストスイッチ発生有無検出の API に関する処理手順を示すフローチャート

【図 9】コンテキストスイッチの処理手順を示すフローチャート

【図 10】コンパクションの処理手順を示すフローチャート

【図 11】複写法による GC の処理手順を示すフローチャート

【図 12】複写法 GC における排他制御の処理手順を示すフローチャート

30 【図 13】複写法 GC における他の排他制御の処理手順を示すフローチャート

【図 14】図 13 における排他制御用 API に関する処理手順を示すフローチャート

【図 15】図 13 における排他制御用 API に関する処理手順を示すフローチャート

【図 16】GC スレッドの優先度の自動切替の例を示す図

【図 17】スレッドの優先度値および優先度時間の切替に関するフローチャート

40 【図 18】GC スレッドの高優先度時間を変更した例を示す図

【図 19】GC スレッドの高優先度時間を変更可能とするためのフローチャート

【図 20】GC スレッドの高優先度時間を自動変更する例を示すフローチャート

【図 21】GC スレッドの高・低優先度時間の切り替え周期を変更した例を示す図

【図 22】GC スレッドの高・低優先度時間の切り替え周期を変更するためのフローチャート

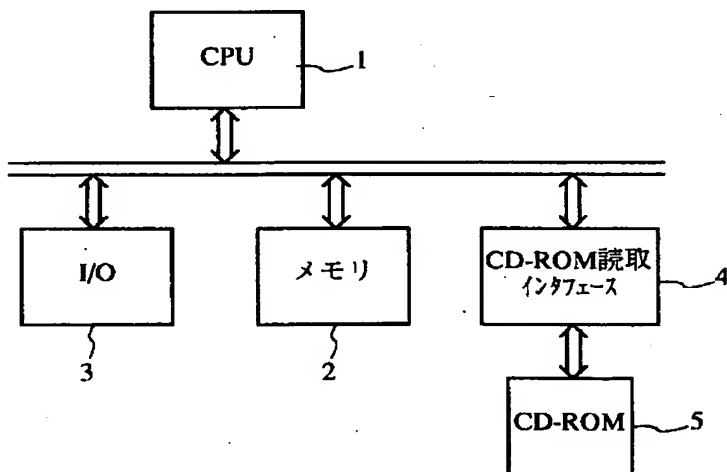
50 【図 23】不定期な GC 処理の例を示す図

- 【図24】図21に対応するフローチャート  
 【図25】排他制御APIによるGCスレッドの優先度の強制変更の例を示す図  
 【図26】排他制御APIによるGCスレッドの優先度を強制変更するためのフローチャート  
 【図27】GCのアルゴリズムを切り替える処理手順を示すフローチャート  
 【図28】生成されるオブジェクトのサイズの分布の例を示す図  
 【図29】オブジェクトのメモリの割り当てサイズの例を示す図  
 【図30】オブジェクト生成の処理手順を示すフローチャート  
 【図31】オブジェクトサイズの分布検知および固定サイズの決定処理に関するフローチャート  
 【図32】ヒープ領域の構成を示す図  
 【図33】オブジェクト消去の手順を示すフローチャート  
 【図34】オブジェクト生成の手順を示すフローチャート  
 【図35】マーク&スイープ法によるGCの手順を示すフローチャート  
 【図36】マーククリアの処理手順を示すフローチャート  
 【図37】オブジェクト消去の処理手順を示すフローチャート

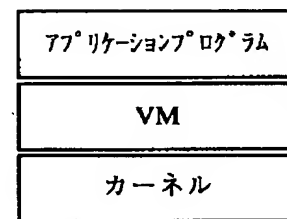
\*

- \* 【図38】ツリー探索によるマーク付与の例を示す図  
 【図39】図38に対応するフローチャート  
 【図40】図38に対応するフローチャート  
 【図41】図38に対応するフローチャート  
 【図42】図38に対応するフローチャート  
 【図43】ツリー探索によるマーク付与の例を示す図  
 【図44】図43に対応するフローチャート  
 【図45】図43に対応するフローチャート  
 【図46】図43に対応するフローチャート  
 【図47】参照変更のマーク付与の他の処理手順を示すフローチャート  
 【図48】オブジェクトに対応するマーク付与の他の処理手順を示すフローチャート  
 【図49】固定サイズの設定とオブジェクト生成に関する処理手順を示すフローチャート  
 【図50】オブジェクトサイズ分布設定に関する処理手順を示すフローチャート  
 【図51】オブジェクトサイズの分布と分割サイズの例を示す図  
 【図52】ヒープ領域を所定サイズで分割する処理とオブジェクト生成に関する処理手順を示すフローチャート  
 【図53】ヒープ領域を所定サイズで分割する処理に関する処理手順を示すフローチャート  
 【図54】従来のマーク&スイープ法によるGCの手順を示すフローチャート

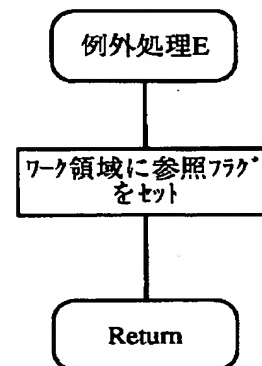
【図1】



【図2】

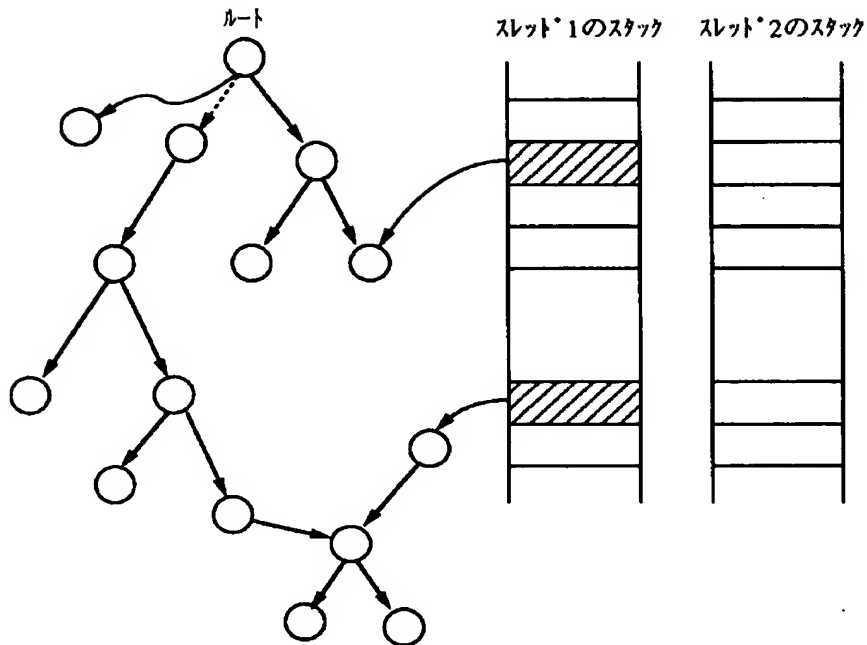


【図15】

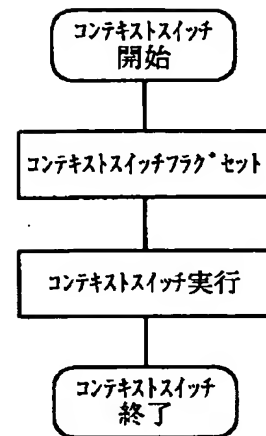




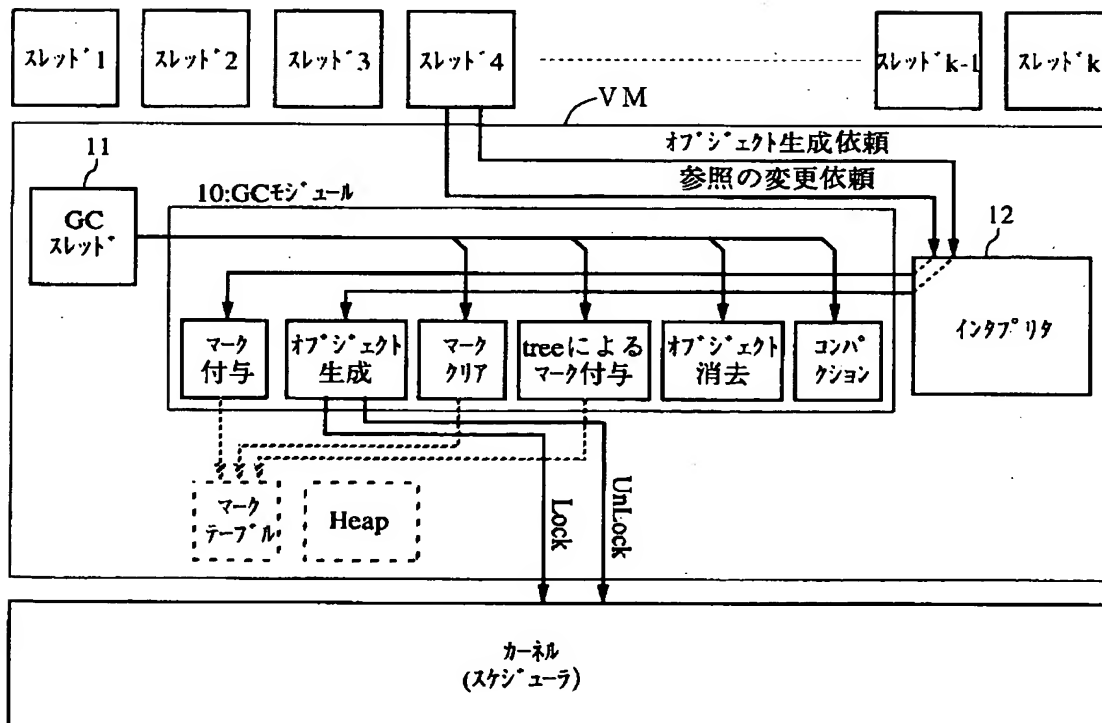
【図3】



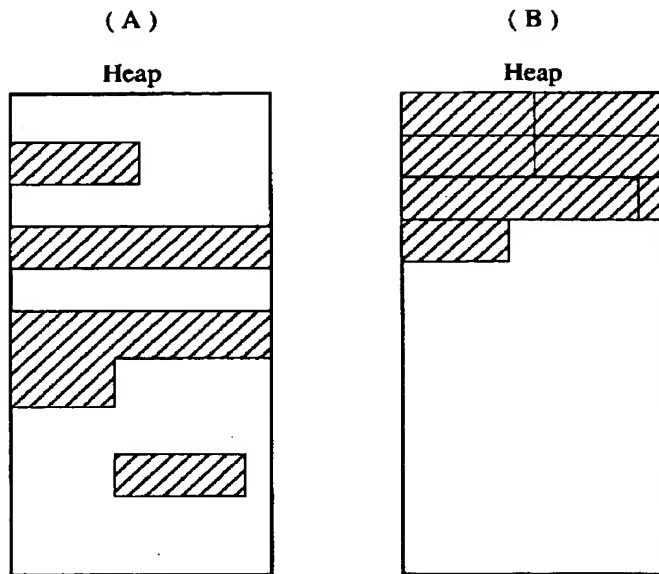
【図9】



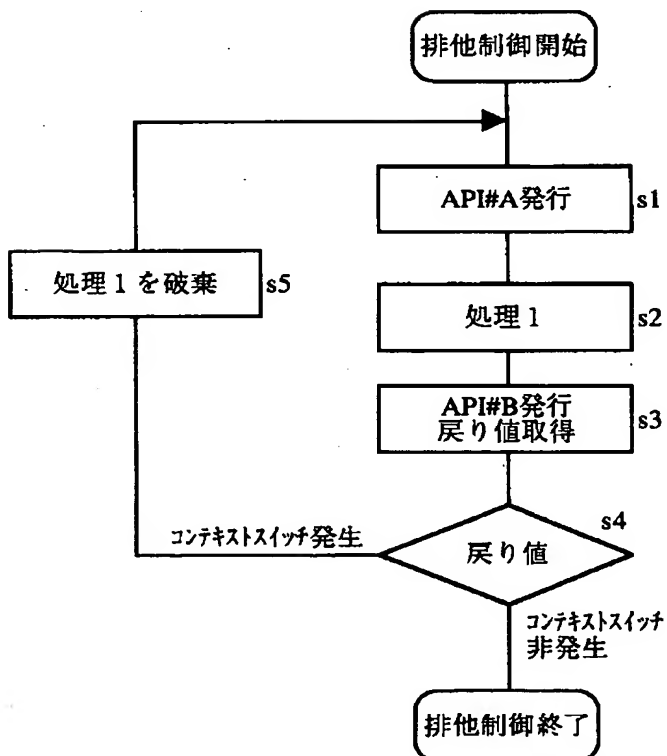
【図4】



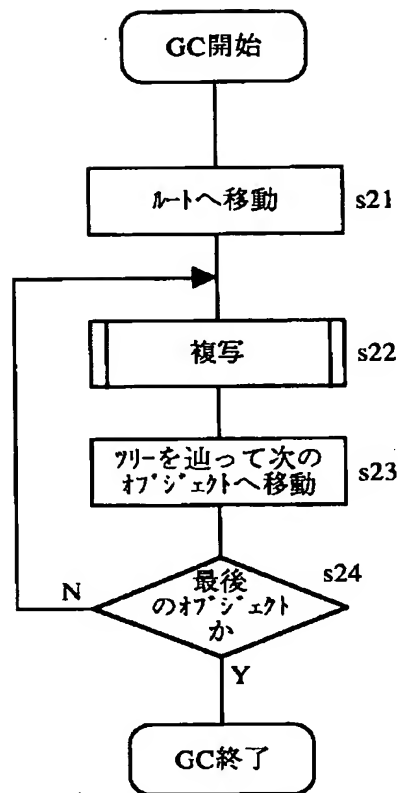
【図5】



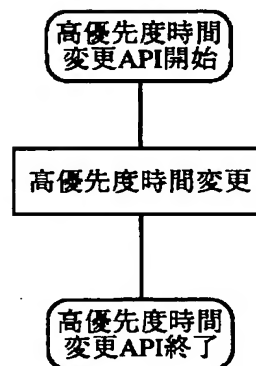
【図7】



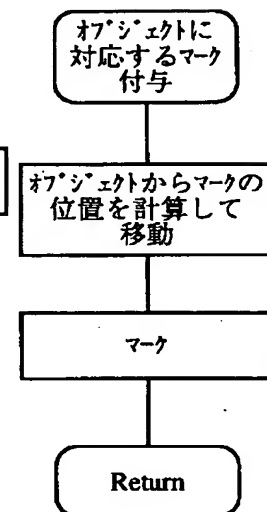
【図11】



【図19】

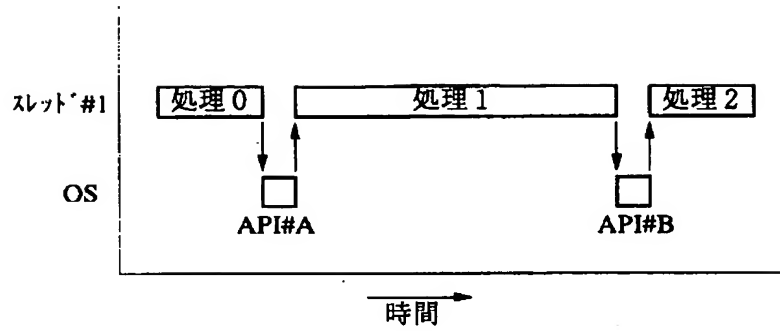


【図46】

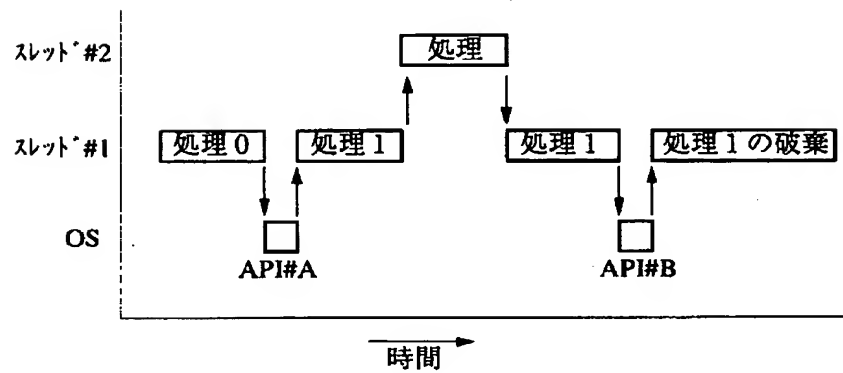


【図6】

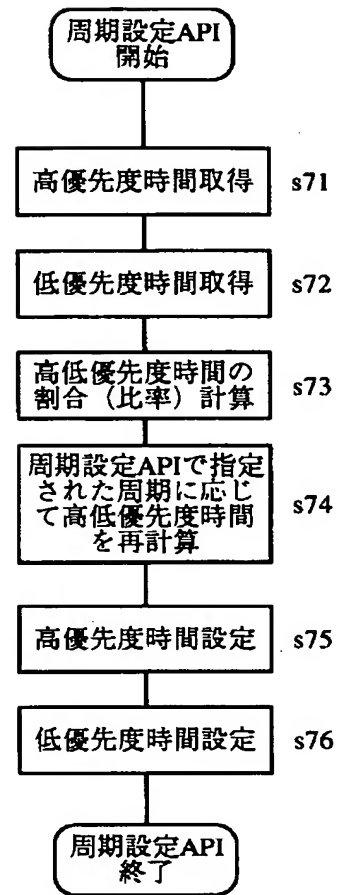
(A)



(B)



【図22】



【図29】

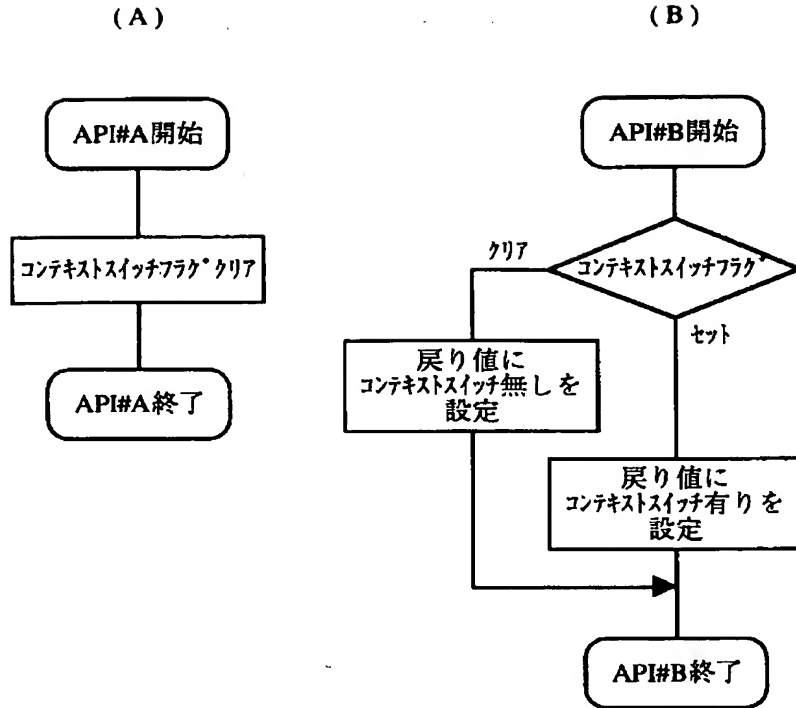
(A)

Obj	Obj	Obj	Obj
Obj	Obj	Obj	Obj
Obj	Obj	Obj	Obj
Obj	Obj	Obj	Obj
Obj	Obj	Obj	Obj

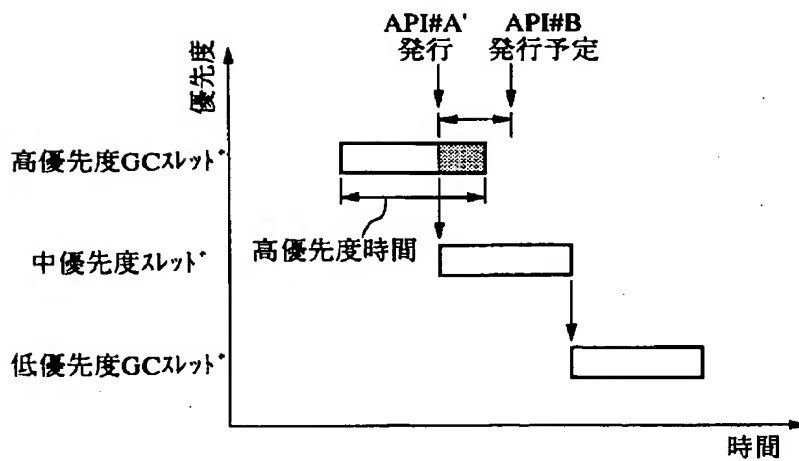
(B)

Obj	Obj	Obj	Obj
Obj	Obj	Obj	Obj
Obj	Obj	Obj	Obj
Obj	Obj	Obj	Obj
Obj	Obj	Obj	Obj

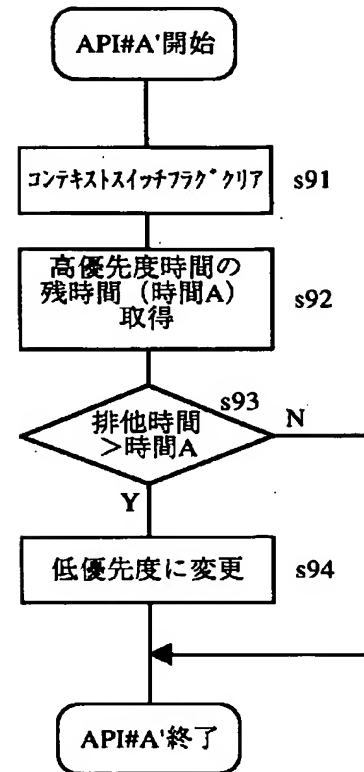
【図8】



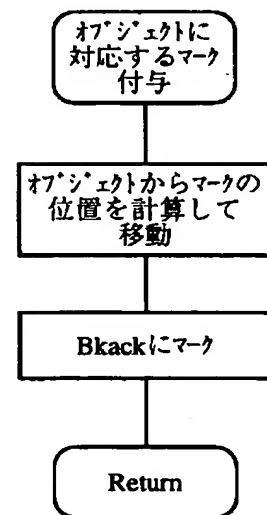
【図25】



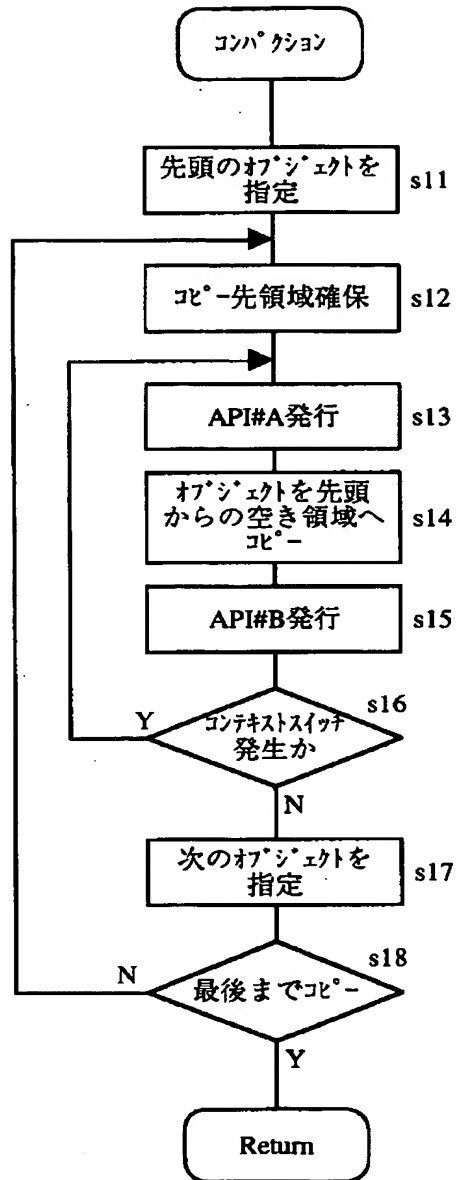
【図26】



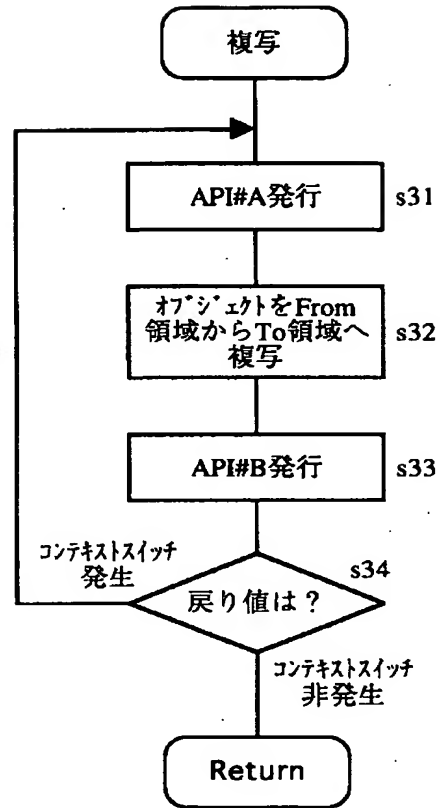
【図48】



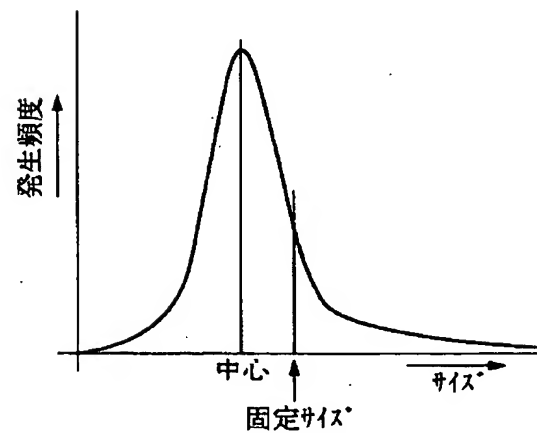
【図10】



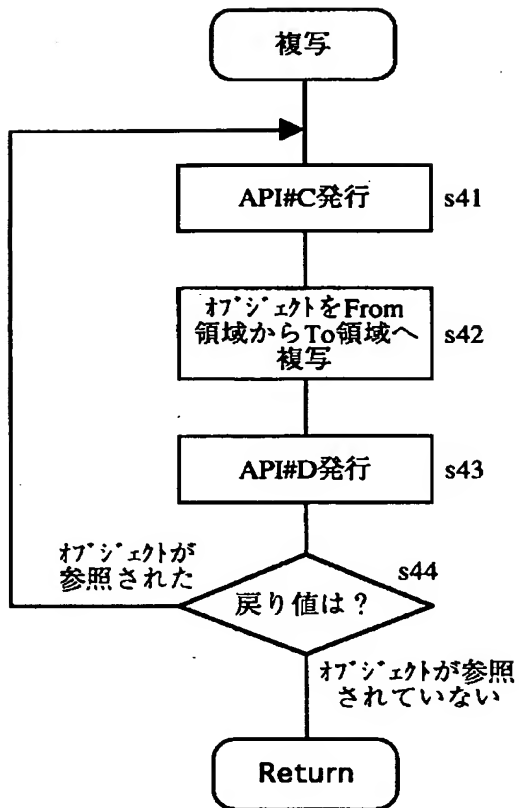
【図12】



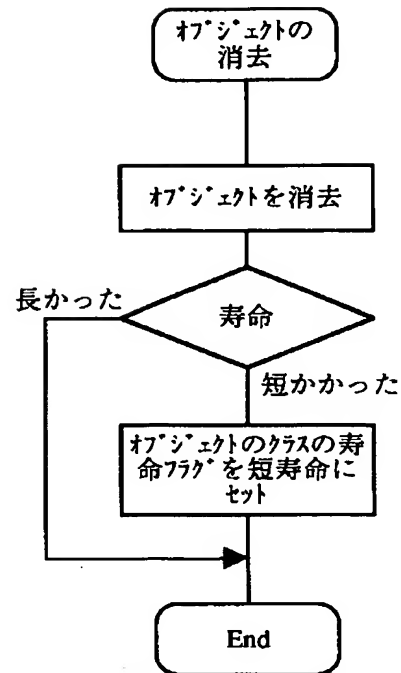
【図28】



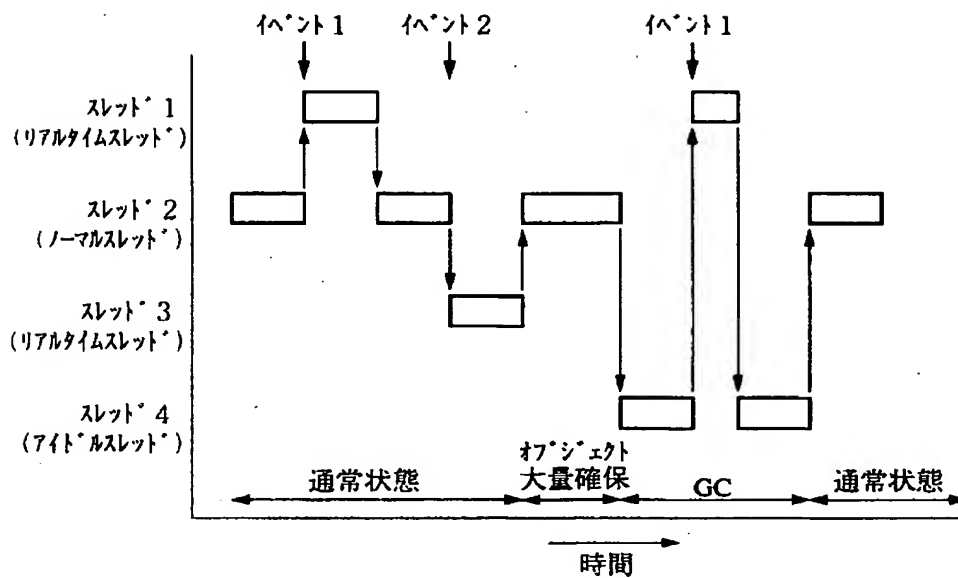
【図13】



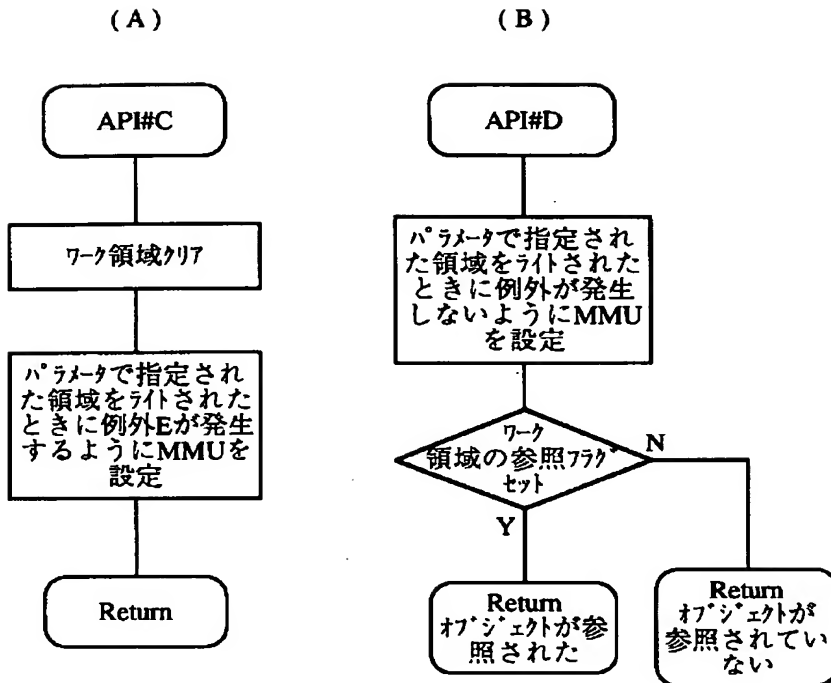
【図33】



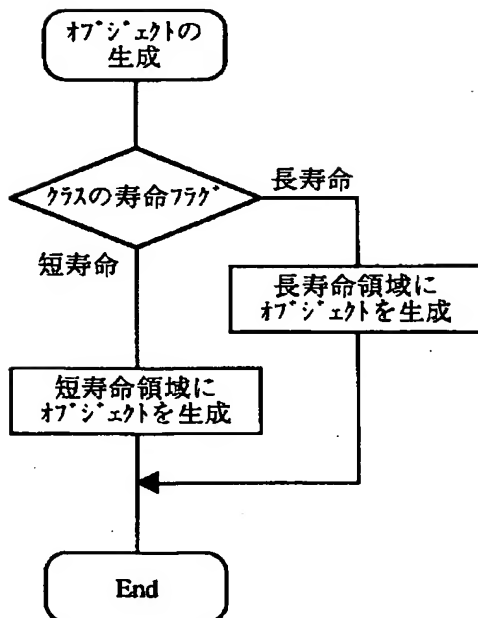
【図23】



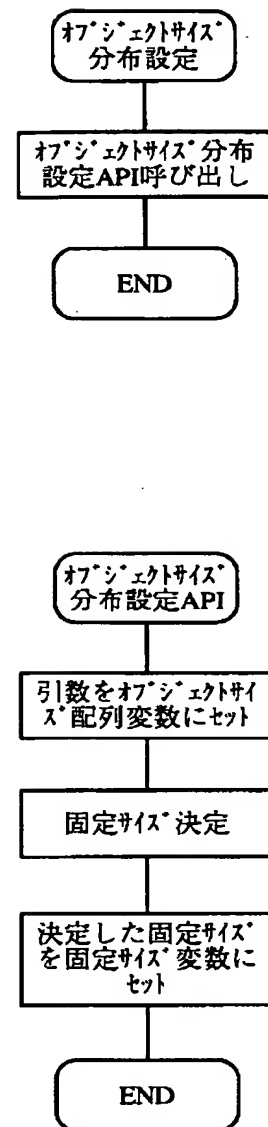
【図14】



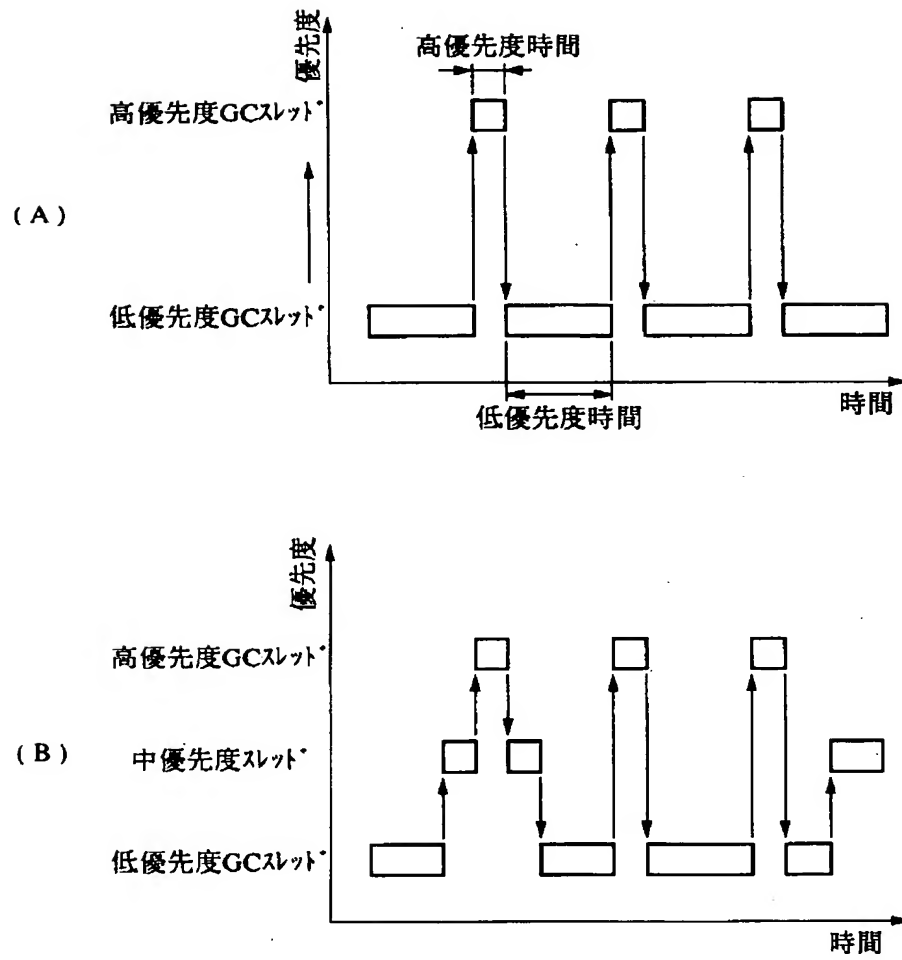
【図34】



【図50】

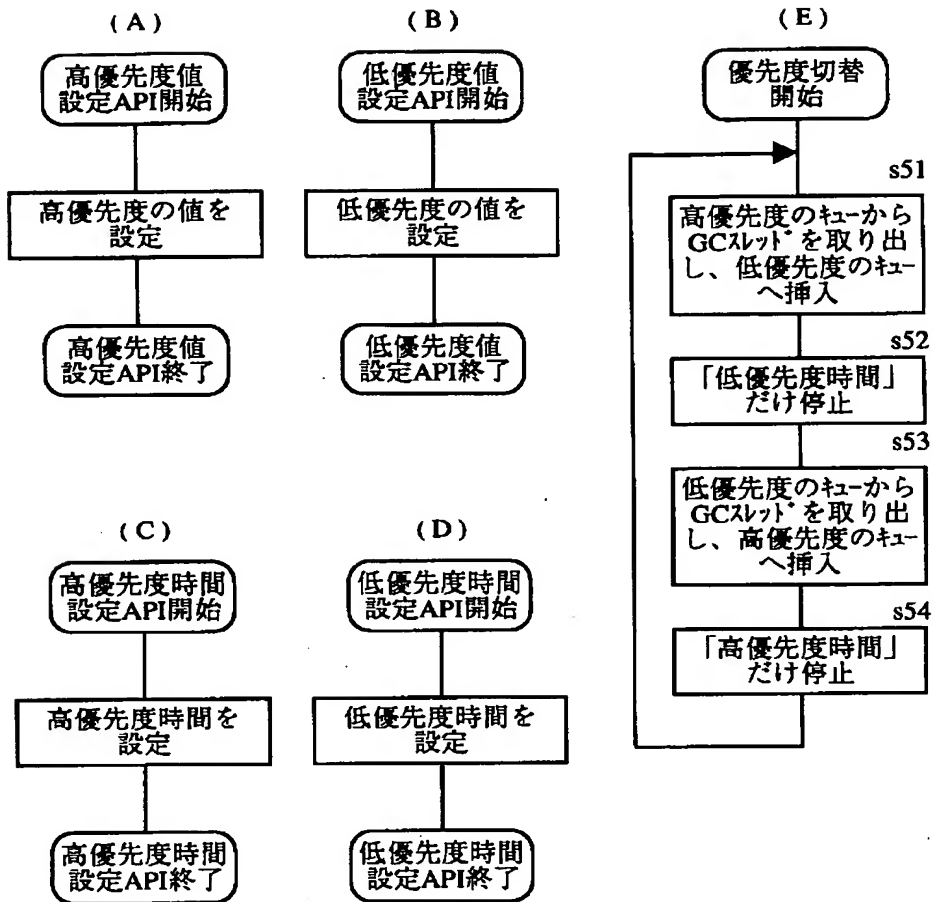


【図16】

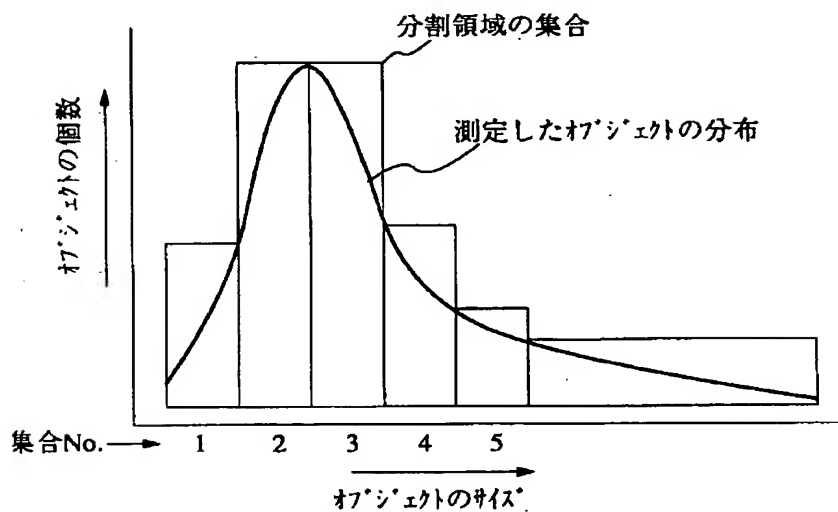




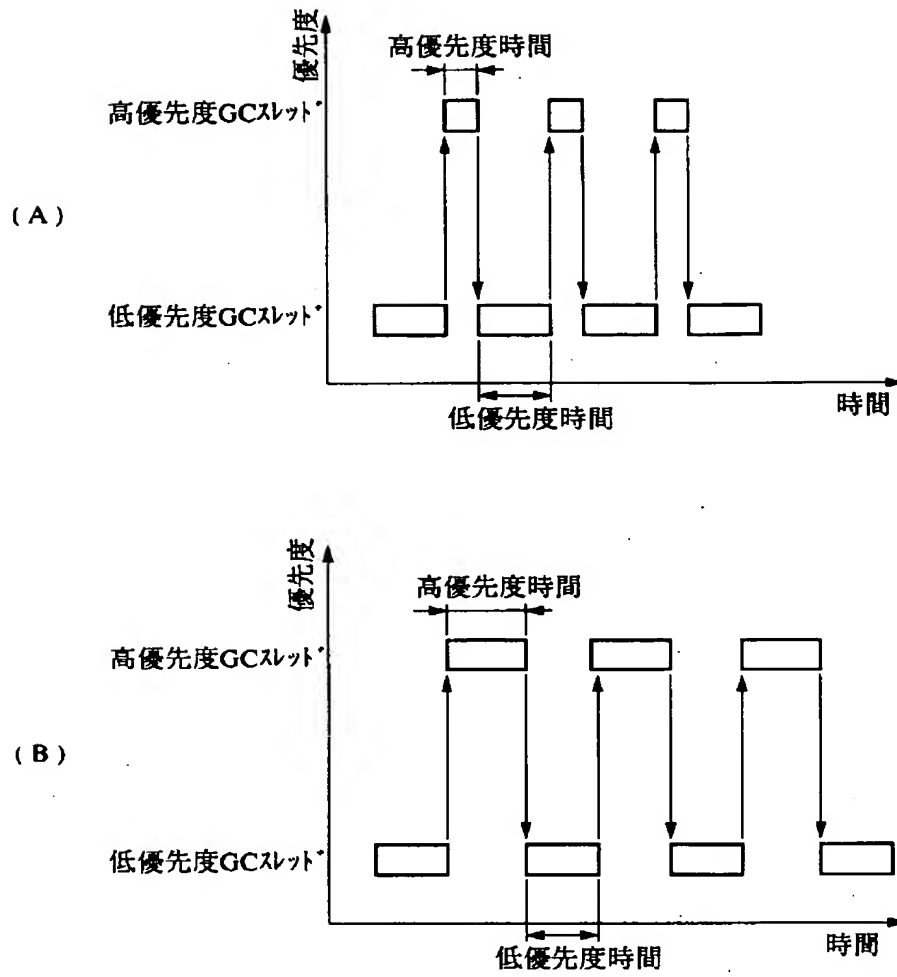
【図17】



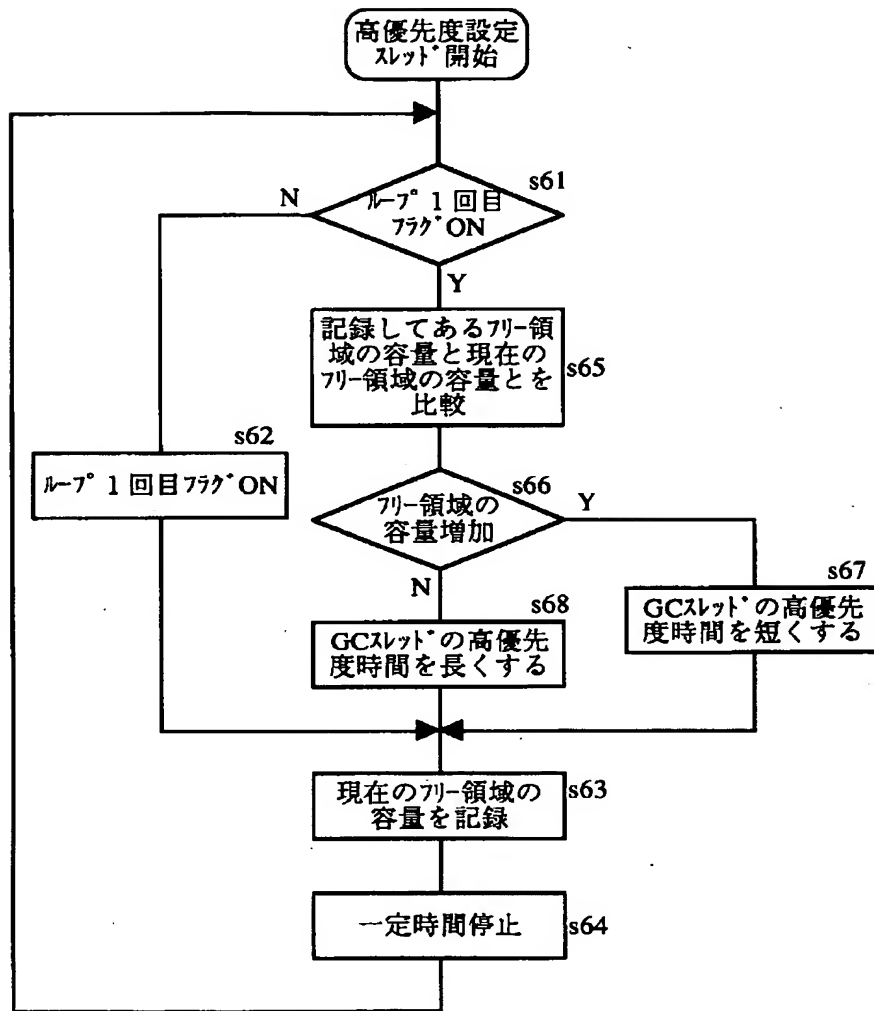
【図51】



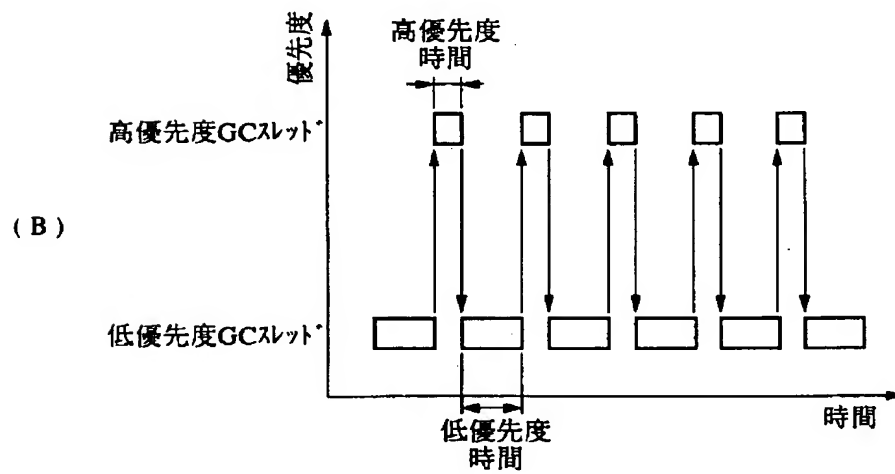
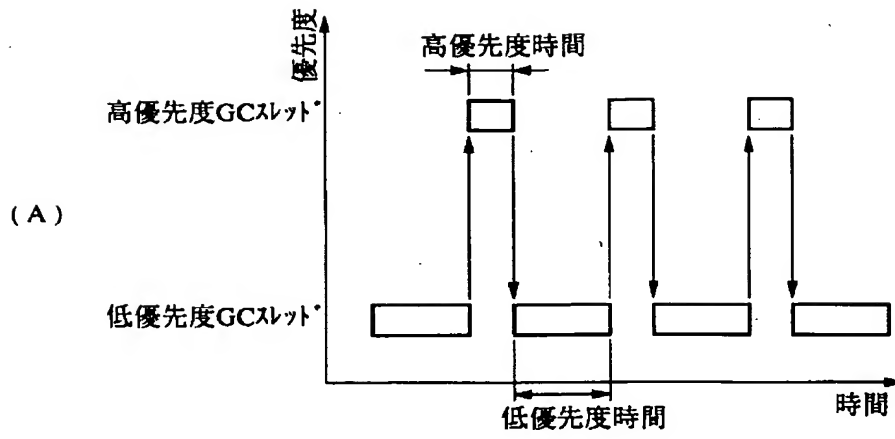
【図18】



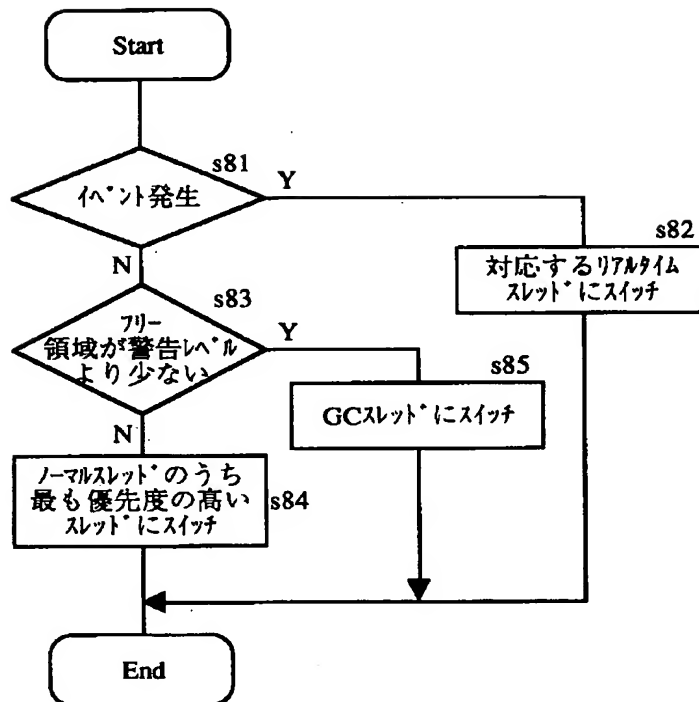
【図20】



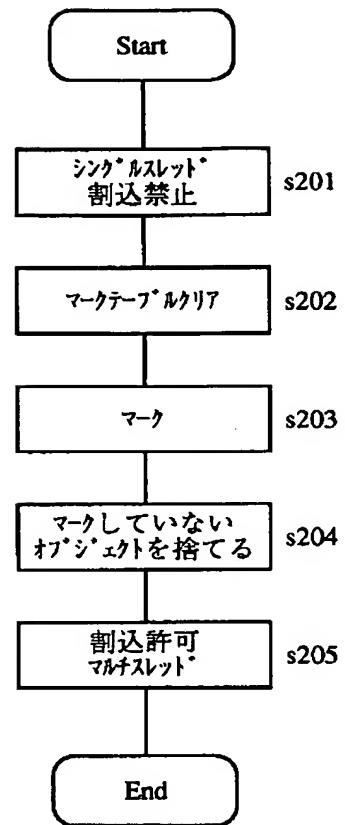
【図21】



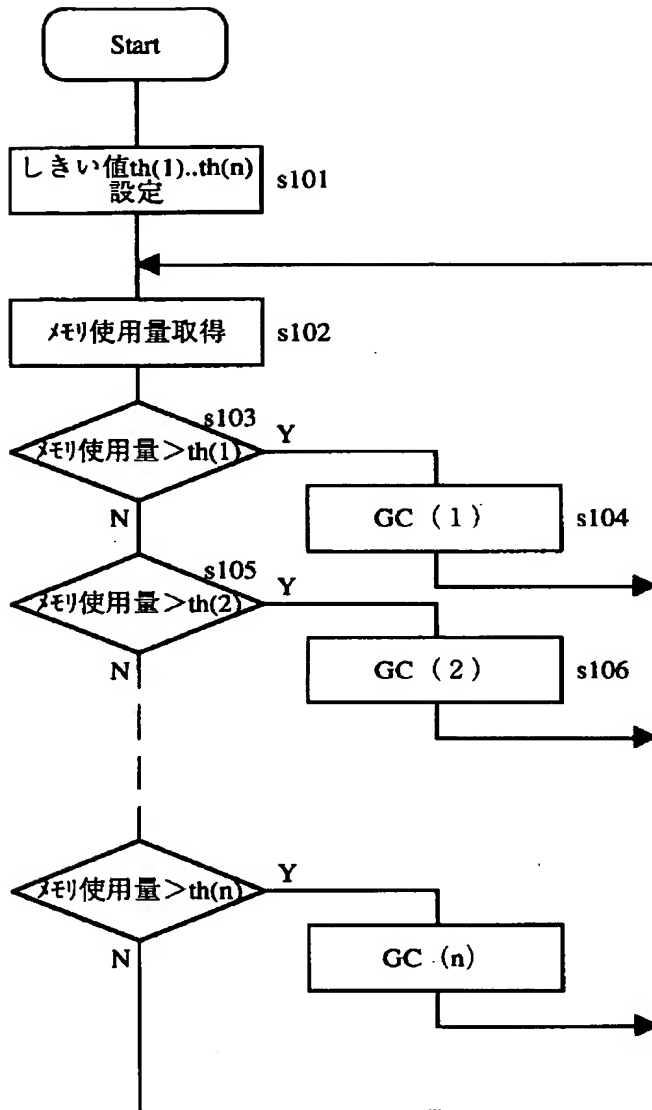
【図24】



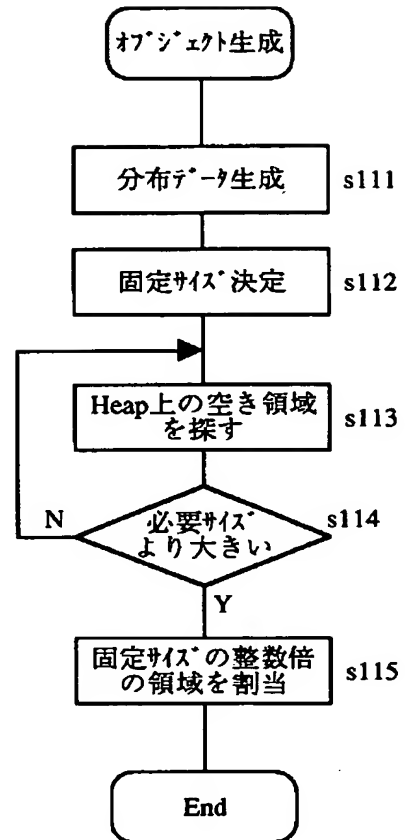
【図54】



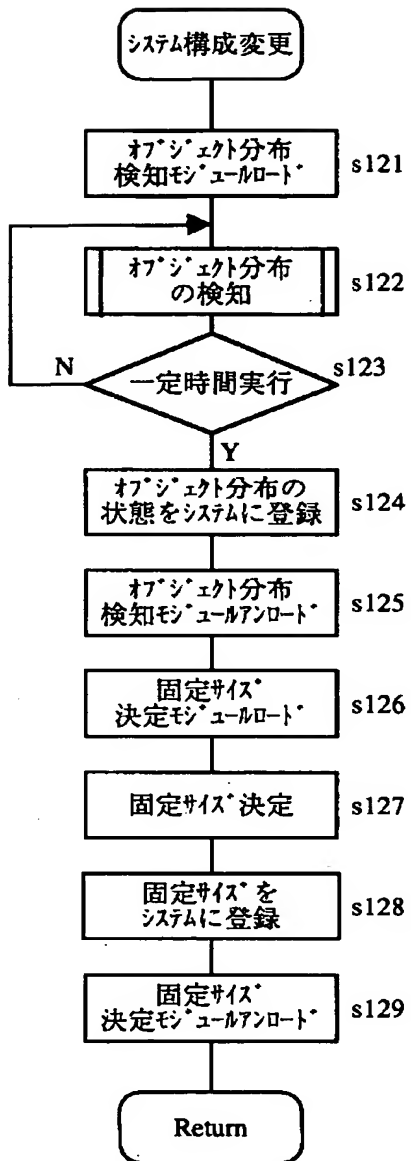
【図27】



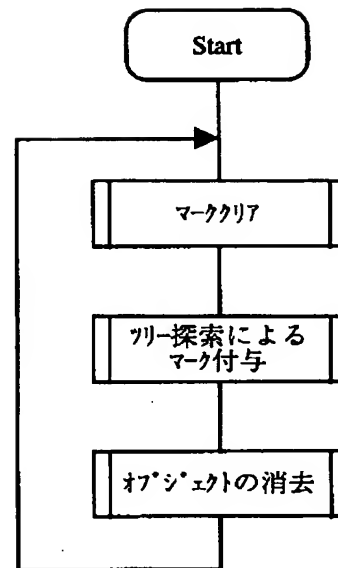
【図30】



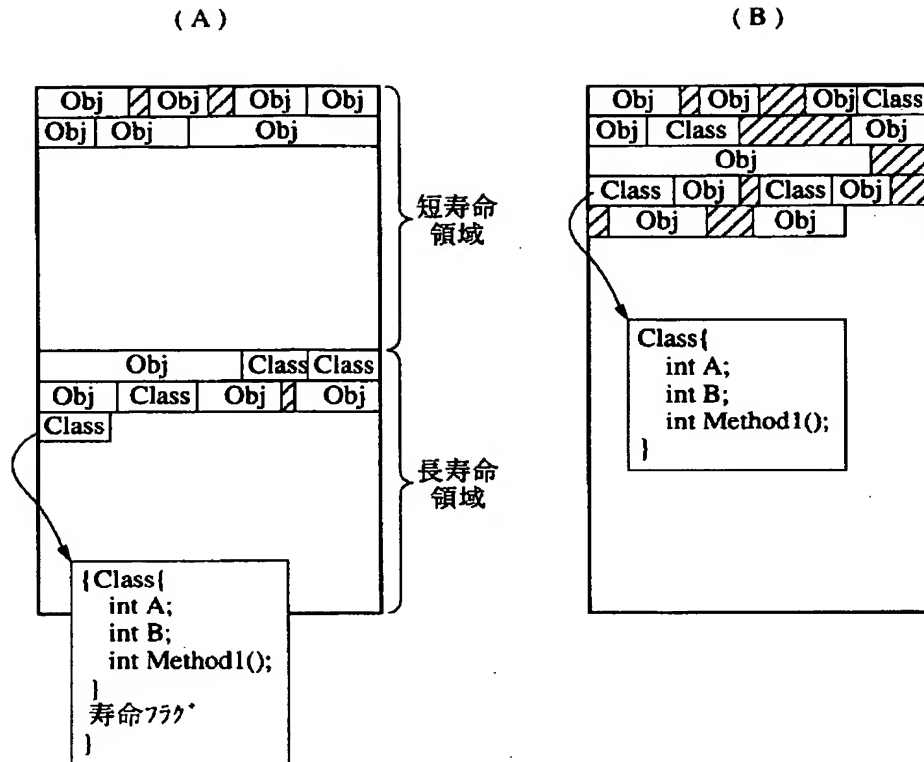
【図31】



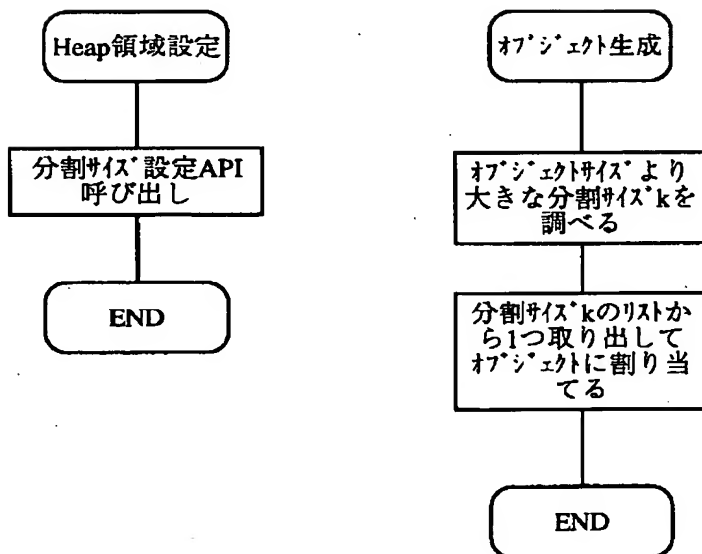
【図35】



【図32】

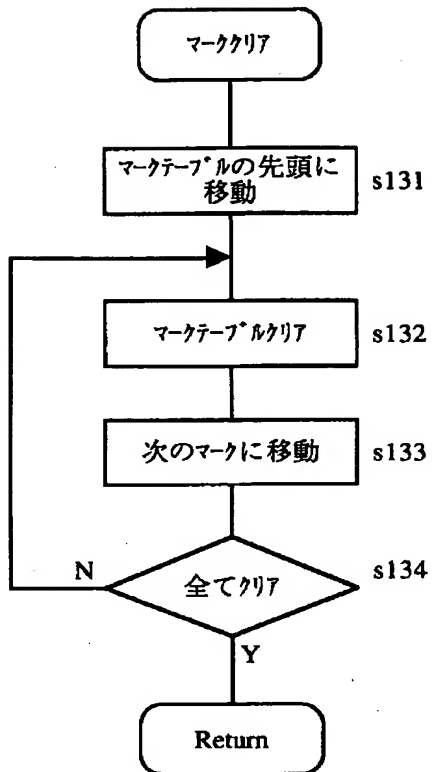


【図52】

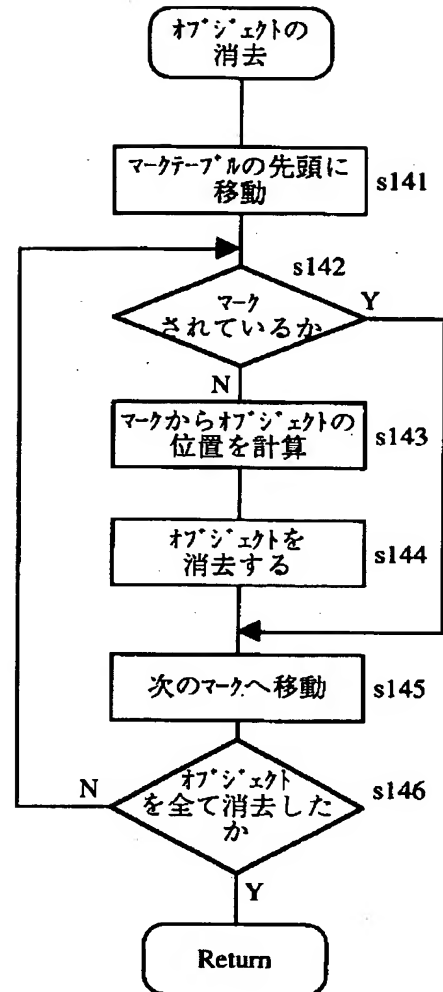




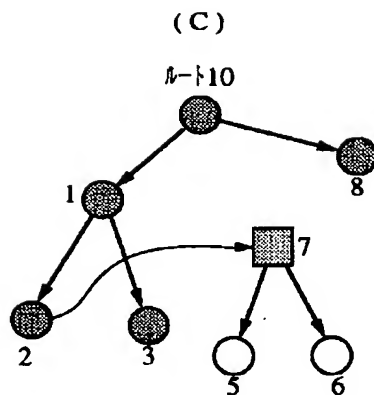
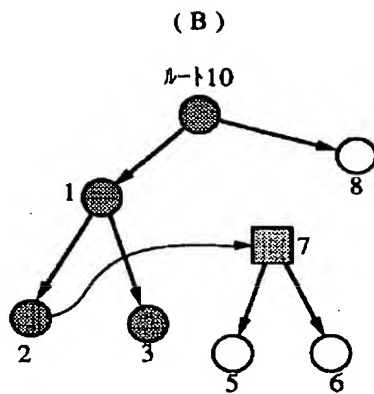
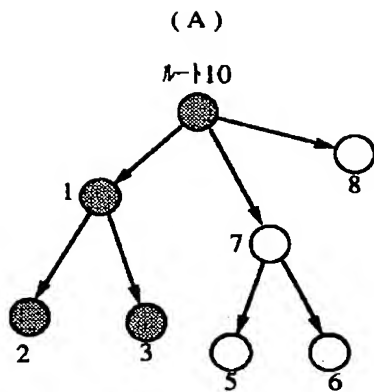
【図36】



【図37】

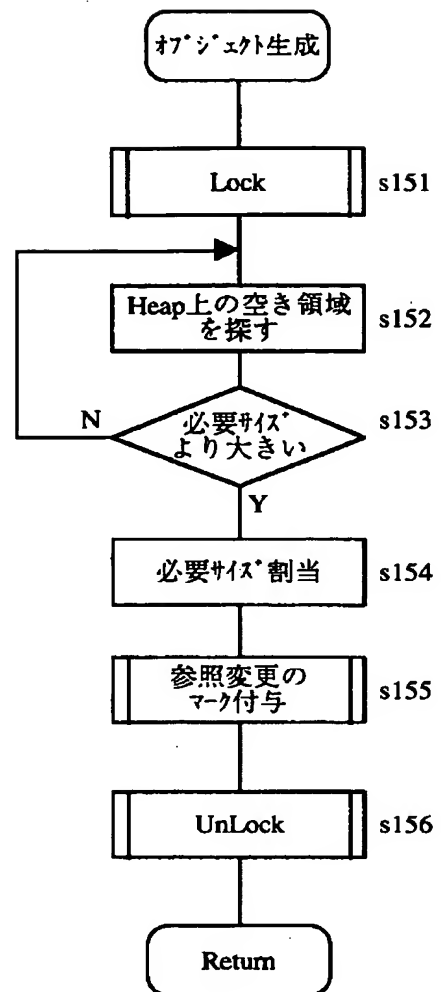


【図38】

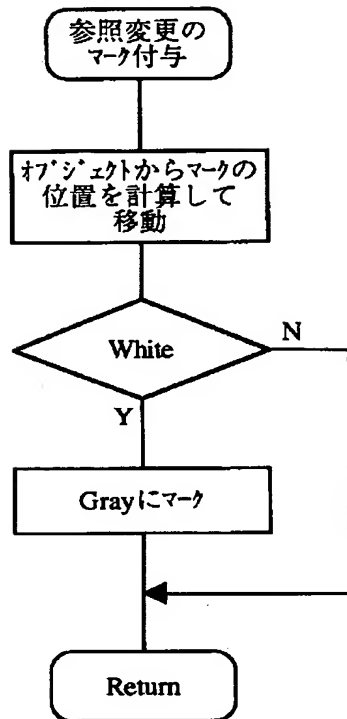


● Black  
○ White  
■ Gray

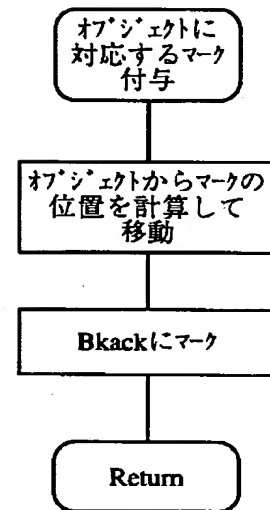
【図39】



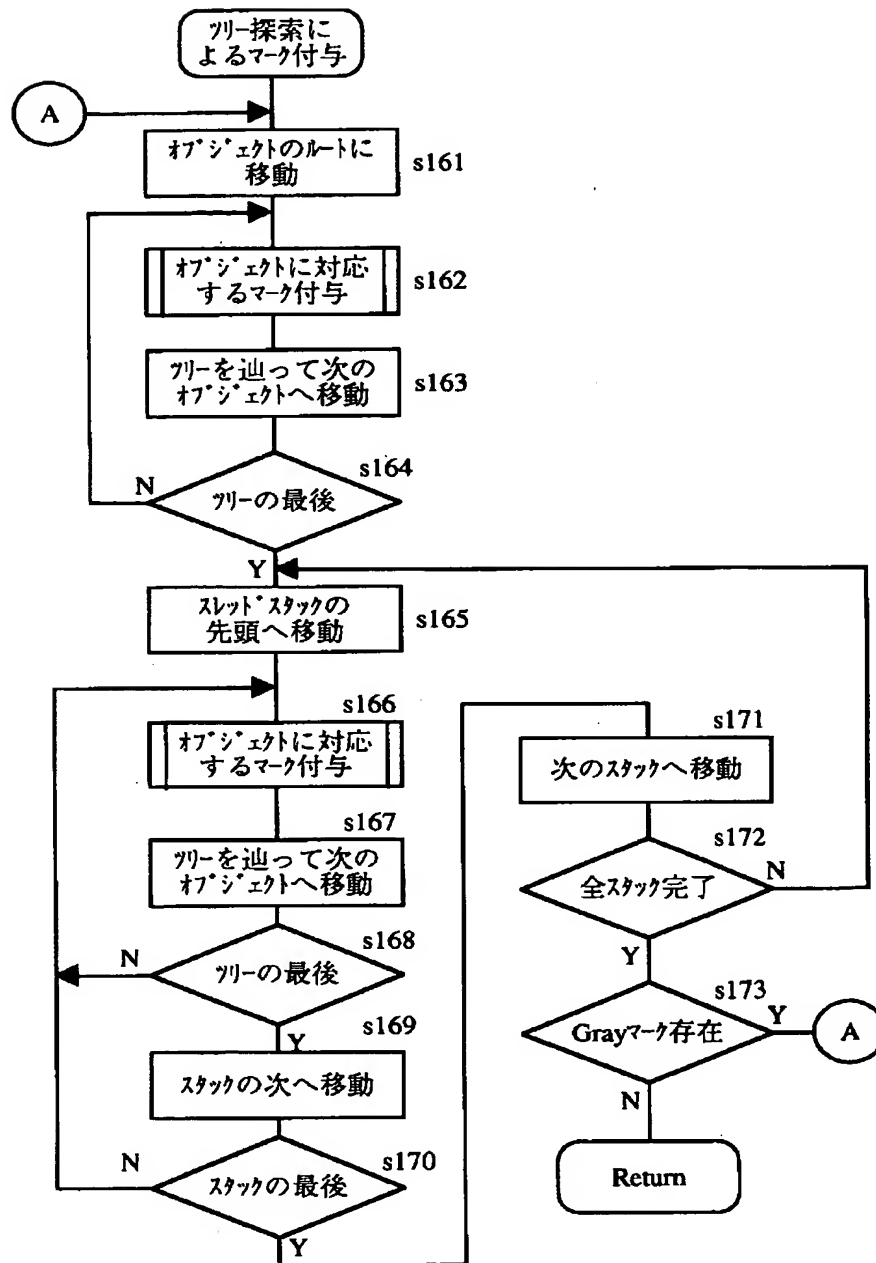
【図40】



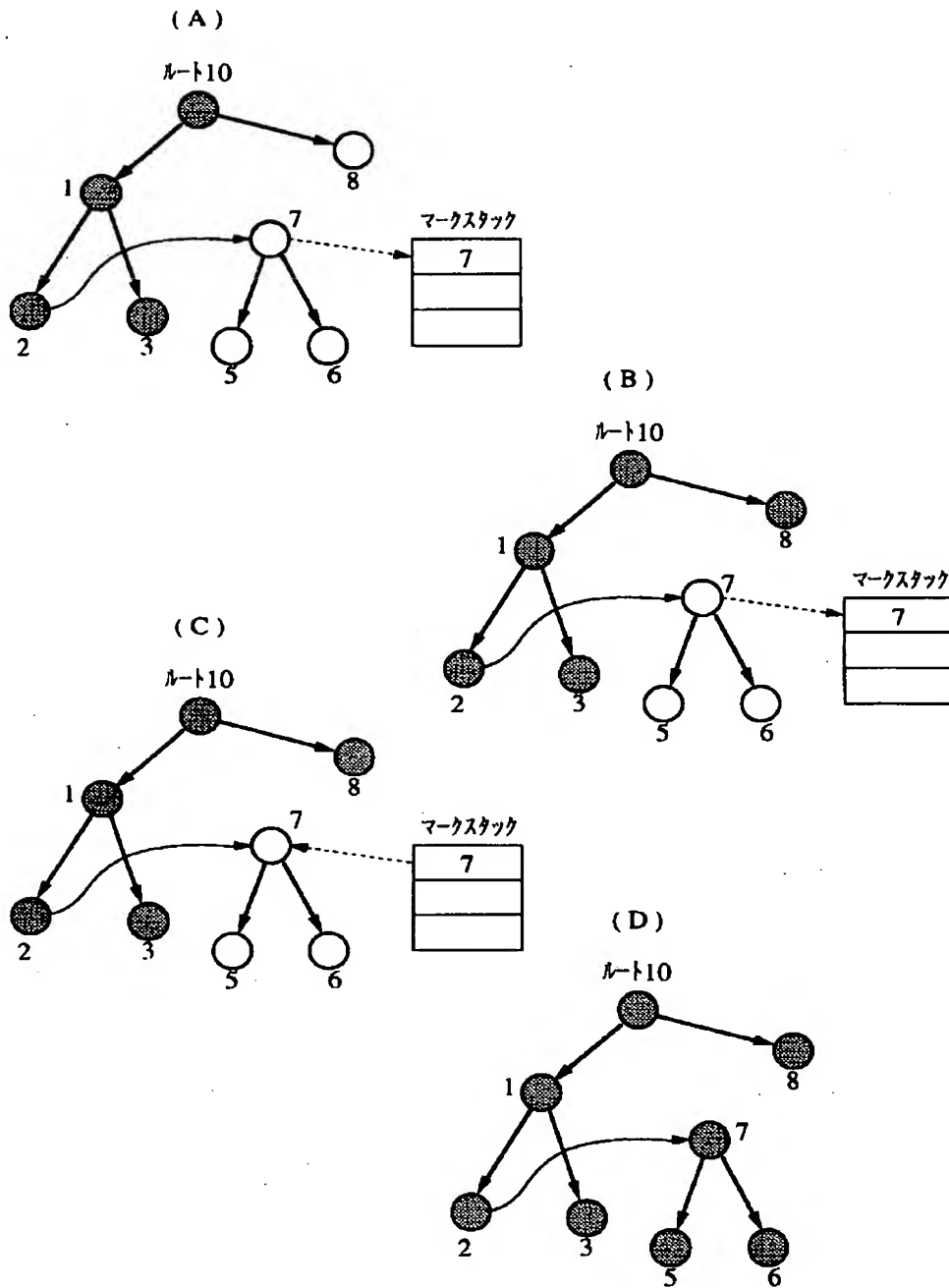
【図42】



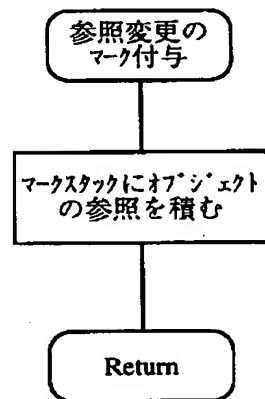
【図41】



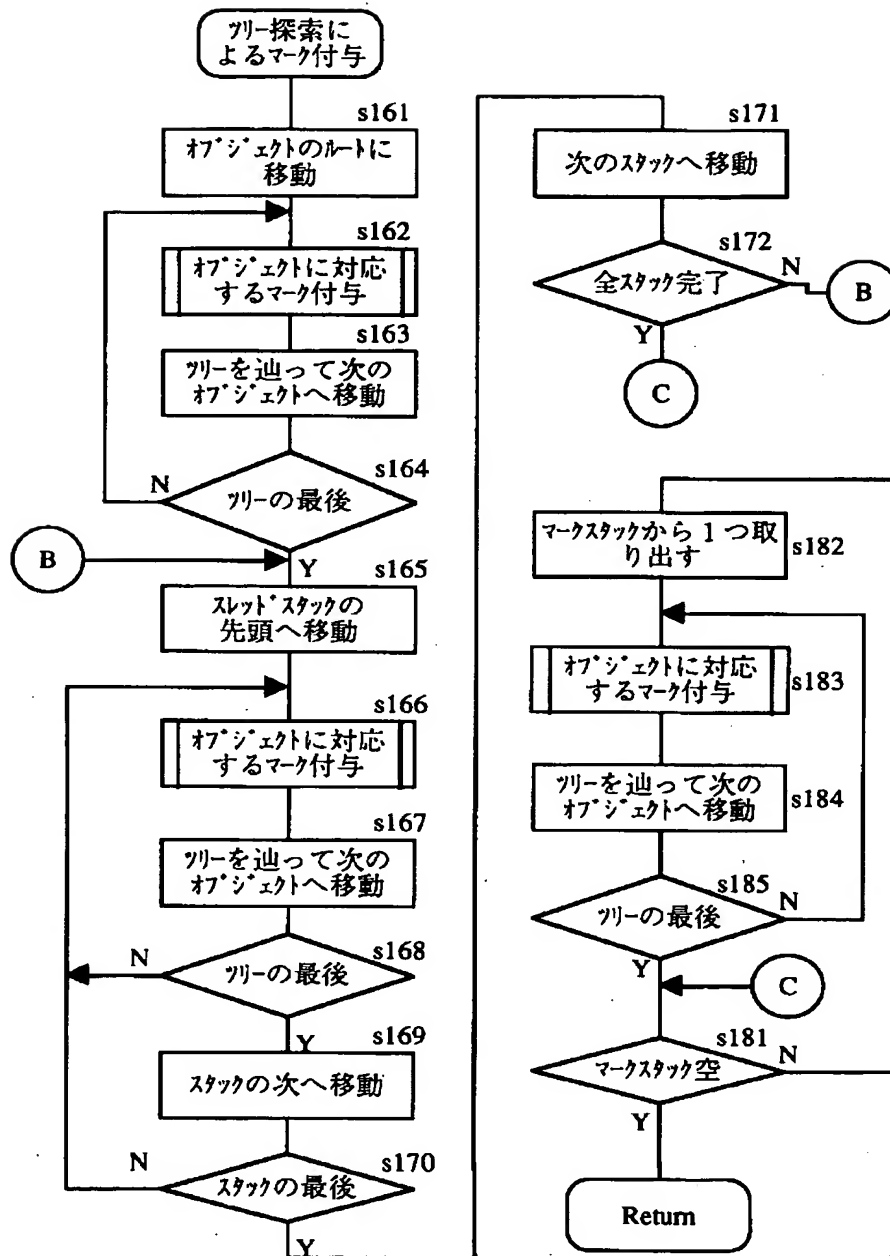
【図43】



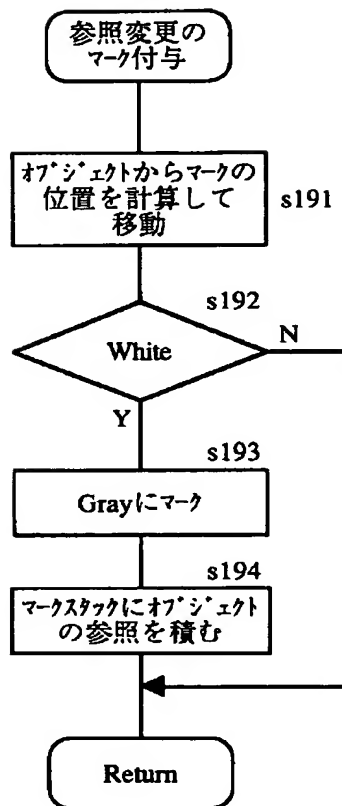
【図44】



【図45】

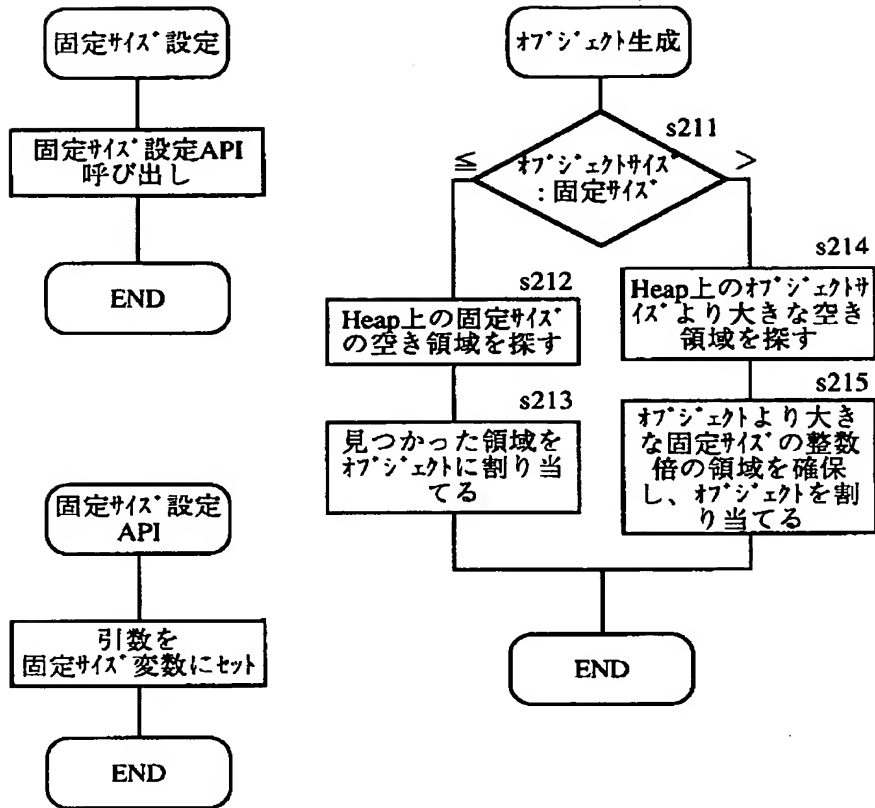


【図47】

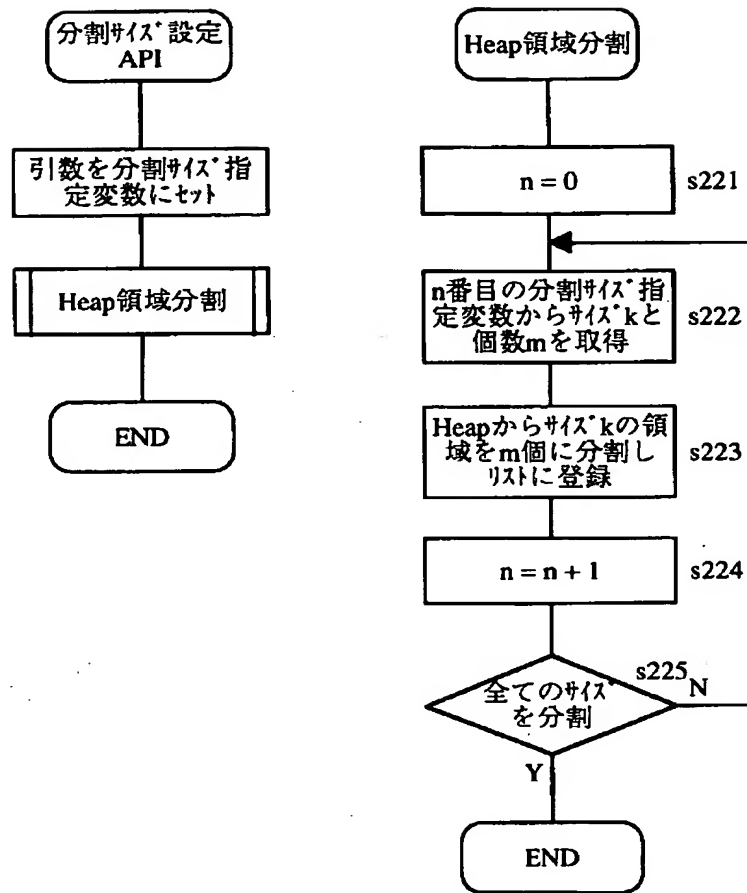




【図49】



【図53】



フロントページの続き

(72)発明者 栗林 博  
京都府京都市右京区花園土堂町10番地  
オムロン株式会社内

(56)参考文献 特開 平1-217638 (J P, A)  
特開 平3-141442 (J P, A)  
特開 平8-153037 (J P, A)  
特開 平6-208502 (J P, A)  
特開 昭61-75649 (J P, A)  
特開 平4-38540 (J P, A)  
発明協会公開技報・公技番号94-6618  
発明協会公開技報・公技番号92-1076

(58)調査した分野(Int.Cl.<sup>7</sup>, DB名)

G06F 9/46

G06F 12/00

G06F 9/44

G06F 12/02

G06F 11/14